A New Searchable Variable-to-Variable Compressor*

Nieves R. Brisaboa¹, Antonio Fariña¹, Juan R. López¹
Gonzalo Navarro², and Eduardo R. López¹
¹ Database Lab, University of A Coruña, A Coruña, Spain.
{brisaboa,fari,mon,erodriguezl}@udc.es
² Dept. of Computer Science, University of Chile, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract

Word-based compression over natural language text has shown to be a good choice to trade compression ratio and speed, obtaining compression ratios close to 30% and very fast decompression. Additionally, it permits fast searches over the compressed text using Boyer-Moore type algorithms. Such compressors are based on processing fixed source symbols (words) and assigning them variable-byte-length codewords, thus following a fixed-to-variable approach.

We present a new variable-to-variable compressor (v2vdc) that uses words and phrases as the source symbols, which are encoded with a variable-length scheme. The phrases are chosen using the longest common prefix information on the suffix array of the text, so as to favor long and frequent phrases. We obtain compression ratios close to those of p7zip and ppmidi, overcoming bzip2, and 8–10 percentage points less than the equivalent word-based compressor. V2vdc is in addition among the fastest to decompress, and allows efficient direct search of the compressed text, in some cases the fastest to date as well.

1 Introduction

The growth of text databases has boosted the interest on new text compression techniques able of considerably reducing their storage size, while retaining the ability of managing such large text databases in compressed form. The rise of the first word-based text compressors [2, 14] showed that compressors using a semistatic zero-order word-based modeler were able to reduce text collections to around 25% of their original size when coupled with a bit-oriented Huffman [8] coder.

Years later, two word-based byte-oriented compressors called Plain Huffman (PH) and Tagged Huffman (TH) were presented [15]. The first one consisted in a word-based modeler coupled with a 256-ary Huffman coder. PH yielded worse compression ratios (around 30%) but gained decoding efficiency. TH was similar, but it reserved 1 bit of each byte to mark the beginning of the codewords. This worsened compression

*This work was funded in part by MCIN grant TIN2009-14560-C03-02 and Xunta de Galicia grant 2006/4 (for the Spanish group), by AECI grant A/8065/07 (all authors), and by Fondecyt grant 1-080019 (fourth author).
ratio to around 33%, but boosted searches over text compressed with $TH$, as Boyer-Moore type searching [3] became possible. In addition, such marks made $TH$ a self-synchronized code, this way enabling random decompression.

Dense codes [4] yield similar features as those of $TH$ while obtaining compression ratios just slightly worse than $PH$. Among such codes, End-tagged dense code (etdc) compresses slightly worse than $(s,c)$-dense code but is a bit faster. The simple encoding and decoding procedures made dense compressors the most relevant word-based ones for natural language text databases.

Yet, the compression obtained by semistatic zero-order byte-oriented compressors is lower-bounded by $PH$, so these techniques cannot compete with powerful compressors such as $p7zip$, $bzip2$, or PPM-based ones [7]. In this paper, we aim at overcoming this limitation by allowing the model to contain not only words, but also phrases (sequences of words). This allows us to represent one whole phrase with just one codeword.

The success of the compressor crucially depends on its ability to choose good phrases. This problem is well known in the more general field of grammar-based compression. As finding the smallest grammar for a text is NP-complete [6] different heuristics exist. Some examples are LZ78 [20], re-pair [11], and Sequitur [17], among many others [1, 18, 6]. For example, re-pair [11] gathers phrases pairwise and recursively: it performs multiple passes over the text, at each pass forming a new phrase with the most frequent pair of symbols. A different approach [1] detects all the non-overlapping phrases in the source data, and uses a gain function to measure the goodness of such phrases.

We follow an approach similar to the latter [1], so that our phrases are flat (do not contain others). We use the longest common prefix information of the suffix array of the text, to choose phrases based on length and frequency, giving preference to longer phrases and setting a frequency or gain threshold. Finally, we apply zero-order modelling and etdc encoding of the sequence of phrases.

## 2 A new technique: v2vdc

Our new compressor, named v2vdc, is basically composed of a phrase-based modeler coupled with an etdc coder. We consider a phrase any sequence of at least two words which appears minFreq or more times in the original text. The modeling phase detects “good” phrases, aided by the use of both a suffix array [12] and a longest common prefix (LCP) structure.

**Compression.** The process consists of the following stages:

- **Parsing and selection of candidate phrases:** In this phase we identify all the candidate phrases in the source text $T$. The result of this phase is a vocabulary of words and a list of candidate phrases.

  We start by identifying the different words and their frequencies in $T$, obtaining an alphabetically ordered vocabulary of words ($V_w$). We use $V_w$ to create a tokenized representation $T_{ids}$ of $T$ where each word is represented by an integer (its id).
The next step involves the creation of a suffix array (SA) over $T_{ids}$, and a LCP structure. The LCP keeps, for each position $2 \leq j \leq |T_{ids}|$ in SA, the length of the longest common prefix, measured in ids, between the suffixes pointed by $SA[j]$ and $SA[j - 1]$. A traversal of LCP gathers all the candidate phrases in $T_{ids}$, with their length and number of occurrences. Every maximal phrase of length $\geq 2$ appearing $\geq \text{minFreq}$ times is a candidate phrase (“maximal” means that a shorter phrase occurring at the same positions as a longer one is not included). The set of candidate phrases corresponds exactly to the suffix tree nodes of $T$ with at least $\text{minFreq}$ occurrences, and hence they are $O(|T_{ids}|/\text{minFreq})$. The resulting list of candidate phrases ($LPh$) is sorted by decreasing length, breaking ties by decreasing frequency.

- Gathering the final phrase-book and producing a phrase-tokenized representation ($T_{ph}$) of $T$. We include both words and phrases in a common phrase-book.

We start with $T_{ph} = T_{ids}$. Then we traverse $LPh$ and, for each candidate phrase $ph_i$, we check its frequency in $T$ counting only the occurrences of $ph_i$ that do not overlap with phrases already included (a bitmap of size $|T_{ids}|$ marks such phrases). If $ph_i$ still deserves to be included in the final phrase-book, we mark its occurrences as “used” in the bitmap, decrease the frequency values in $V_w$ of the words it contains, and replace the phrase occurrences in $T_{ph}$. Otherwise, we try successive shorter prefixes of $ph_i$ before finally discarding it (hence the total time can be as high as the number of suffix trie nodes of $T$, yet this is unlikely in practice).

The simplest condition to accept or discard $ph_i$ depends only on whether $\text{freq}(ph_i) \geq \text{minFreq}$. A more sophisticated heuristic estimates the gain in compression obtained by including $ph_i$. This is computed as $(\text{bytes}_{\text{before}} - \text{bytes}_{\text{after}}) * (\text{freq}(ph_i) - 1) - 2$, where: a) $\text{bytes}_{\text{before}} = \sum_j |C_{w_j}|$ is the size of the codewords for the single words $w_j$ that appear in $ph_i$, when $ph_i$ is discarded; b) $\text{bytes}_{\text{after}} = |C_{ph_i}|$ assuming $ph_i$ is accepted; and c) $-1$ and $-2$ are related to the cost of adding $ph_i$ to the phrase-book (see next). To enable the estimation of the value $|C_z|$, the phrase-book is kept sorted by frequency, as in previous work [5].

- Coding and codeword replacement. We use the etdc coder to give all symbols a codeword. Semistatic dense codes [4], and in particular etdc, use a simple encoding scheme that marks the end of each codeword (1 bit from each byte is reserved for this) and assign the 128 most frequent words a 1-byte codeword, the next 128² most frequent words a 2-byte codeword, and so on. Therefore, the codeword length depends only on the range of positions the symbol belongs in the vocabulary sorted by frequency (1..128, 128 + 1..128 + 128², etc.). This permits simple on-the-fly $C_i = \text{encode}(i)$ and $i = \text{decode}(C_i)$ procedures to be performed in $O(\log(i)) = O(|C_i|)$ time [4].

Hence, we sort the phrase-book by frequency to know the number of 1-byte codewords, 2-byte codewords, etc. that will be used for words and for phrases. Prior to encoding, each range within the same codeword length is reorganized (for practical issues explained later): we move words to the beginning and phrases to the end, and finally sort those phrases according to their first occurrence in $T_{ph}$.

Next, encoding of words and phrases is done. We traverse $T_{ph}$ and replace each $id = T_{ph}[i]$ by its corresponding codeword to obtain the compressed data. The only exception to this is that the first occurrence of a phrase is not encoded by its codeword,
but by the sequence of codewords of the words it contains.

Finally, we include the phrase-book in a header stored along with the compressed text. Words are stored explicitly. For phrases, since they appear in the compressed text, we just keep the offset of their first occurrence and their length (in words). We also include the number of words (and phrases) in each range (1-byte, 2-bytes, etc.), which amounts to a few integers.

In order to save space, we compress the sequence of words with p7zip. The phrase offsets in each range (increasingly sorted before the encoding step) are represented differentially, and gaps are encoded with Rice codes [19]; whereas the phrase lengths are encoded with bit-oriented Huffman.

**Decompression.** We start by recovering the phrase-book, and the sequence of plain-text words, \(vocWords\). For each phrase-book entry we keep a pair \(\langle ptr, length \rangle\). For words, those values keep the offset to the corresponding position in \(vocWords\) and the length of the word (in characters). In the case of phrases, that pair contains initially the offset of the first occurrence of the phrase in the compressed text, and the number of words in it. Later, after the first occurrence of the phrase is decoded, this pair will keep an offset to its uncompressed text, and its length (in characters).

As the phrase-book is read, an auxiliary array \(offsets\) is built. For each phrase at position \(id\) in the phrase-book, and appearing first at position \(p\) in the compressed text, \(offsets\) contains an entry \(\langle id, p \rangle\). Then \(offsets\) is sorted increasingly by component \(p\). Note that such sorting is very simple, as the phrases encoded with 1 byte, 2 bytes, etc., were already stored ordered by offset in the header.

Decompression traverses the compressed data (decoding one codeword at a time) and uses \(offsets\) to know the offset where the first occurrence of a new phrase is reached. Basically, the decompressor works as in etdc, by applying \(id = decode(C_{id})\), and outputting the text at \(phrase-book[id].ptr\). However, each time decoding reaches the offset pointed by the next entry \(j\) in \(offsets\), thus detecting the first occurrence of the phrase \(x=offsets[j].id\), it has to decode the next \(phrase-book[id].length\) codewords that make up such phrase. Finally, \(x\) is output, \(j\) is increased, and \(phrase-book[id]\) is updated properly.

Random decompression can be provided by just checking whether \(id = decode(C_{id})\) is a word or phrase (this is very simple as we store in the header the ranges of words and phrases encoded with 1 byte, 2 bytes, etc.). When \(id\) belongs to a phrase, we decompress it by accessing \(phrase-book[id].ptr\) and decoding the following \(phrase-book[id].length\) words from there on. In this case we do not change \(phrase-book[id]\), as the text is supposed to stay in compressed form.

**Searches.** In semistatic word-based compression, direct searches on the compressed text are possible by just compressing the pattern and searching for its compressed form. In a variable-to-variable context, when we search for a single word we need not only to find its codeword, but also the codewords of all phrases containing that word. Additionally, when searching for phrase patterns, we must find also the codes of all the compatible phrases enclosing a substring of the search pattern, as these phrases
can be combined with other words or phrases to make up the full pattern.

Our search algorithm is based on Set-Horspool [16]. When we search for a single-word pattern \( P \), we initially include its codeword \( C_P \) in the Horspool search trie. Later, each time a match occurs we report it, and also check whether such occurrence of \( C_P \) appears inside the first occurrence of a phrase \( ph \) (in this case, we must add \( C_{ph} \) to the search trie). For this sake, we advance in the text (with Horspool) and simultaneously in offsets, as for decompression.

Phrase searches are solved by searching for their least frequent word \( w_{lf} \) in the compressed text. Again, after any match we must check if it is within the first occurrence of a new phrase \( ph \). If so, and if \( ph \) is compatible with the pattern, we add its codeword \( C_{ph} \) to the search trie. For each pattern included in the search trie we store: a) its codeword; b) the number of words missing to its left and right to complete the whole phrase-pattern; and c) the number of times the searched pattern occurs inside \( ph \). To report occurrences we would also have to keep the relative offsets of the pattern within \( ph \).

Each time a codeword \( C_i \) in the search trie is matched, we add up the number of times it contains \( P \). Also, we check the codewords missing both before and after \( C_i \) (as indicated for that entry), looking for a new pattern occurrence to count.

### 3 Experimental results

We used a large text collection from TREC-2: Ziff Data 1989-1990 (ZIFF), as well as two medium-size corpora from TREC-4, namely Congressional Record 1993 (CR) and Financial Times 1991(FT91). As a small collection we used the Calgary corpus\(^1\).

We compared our new compressor using the minFreq threshold method (\( v2vdc \)) as well as the more complex gain heuristic (\( v2vdc_H \)), against well-known compressors such as etdc (http://rosalia.dc.fi.ude.es/codes), gzip (www.gnu.org), 7zip (www.7-zip.org), bzip2 (www.bzip.org), re-pair (www.cbrc.jp/~rwan/software/restore.html, coupled with a bit-oriented Huffman, www.cs.mu.oz.au/~alistair/mr_coder), and ppmdi (http://pizzachili.dcc.uchile.cl, default options).

We provide comparisons on compression ratio, as well as on compression and decompression speed. In addition, to show the performance of the searches over text compressed with \( v2vdc \) we also include experiments performed in both compressed and decompressed text with different search techniques.

Our machine is an Intel Core2Duo E6420@2.13Ghz, with 32KB+32KB L1 cache, 4MB L2 cache, and 4GB of DDR2-800 RAM. It runs 64-bit Ubuntu 8.04 (kernel 2.6.24-24-generic). We compiled with gcc version 4.2.4 and the options -O9 -m32. Time results measure CPU user time.

**Tuning parameter minFreq.** We first focus on how the compression obtained with \( v2vdc \) depends on parameter \( minFreq \) that is, the minimum number of occurrences for a phrase to be considered as a candidate. Figure 1 shows the compression

---

\(^1\)ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus
Figure 1: Compression ratio depending on the parameter \( \text{minFreq} \). The plots named “comp. seq.” refer to the the size of the compressed text, whereas those named “total file” show the overall compression as they include also the size of the header.

The ratio obtained depending on \( \text{minFreq} \) for our corpora, and considering both \texttt{v2vdc} and \texttt{v2vdc}_H. It can be seen that, in general, \( \text{minFreq} \in [5\ldots10] \) leads to good compression, and that the more complex heuristic performs better.

Table 1 shows, for corpus CR, the number of phrases that are gathered during the modeling phase, for the three codeword-length ranges (1, 2, and 3 bytes) that are used by the \texttt{etdc} coder in \texttt{v2vdc} and \texttt{v2vdc}_H. On the one hand, when high \( \text{minFreq} \) values are set, we favor the inclusion of phrases that will probably lead to a large gain in compression. However, the number of phrases occurring many times is not so high, and consequently compression cannot benefit from the gain provided by less frequent phrases. For example, in corpus CR, there are only around 15,500 phrases that occur at least 50 times, whereas around 200,000 phrases occur more than 5 times.

<table>
<thead>
<tr>
<th>minFreq</th>
<th>\texttt{v2vdc}</th>
<th>\texttt{v2vdc}_H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Codeword length (bytes)</td>
<td>Number of phrases</td>
</tr>
<tr>
<td>2</td>
<td>1 2 3</td>
<td>11 6,452 617,388</td>
</tr>
<tr>
<td>3</td>
<td>1 2 3</td>
<td>18 6,742 370,156</td>
</tr>
<tr>
<td>5</td>
<td>1 2 3</td>
<td>19 7,228 193,662</td>
</tr>
<tr>
<td>6</td>
<td>1 2 3</td>
<td>19 7,290 154,957</td>
</tr>
<tr>
<td>7</td>
<td>1 2 3</td>
<td>19 7,464 129,487</td>
</tr>
<tr>
<td>10</td>
<td>1 2 3</td>
<td>22 7,709 83,637</td>
</tr>
<tr>
<td>50</td>
<td>1 2 3</td>
<td>30 8,788 6,711</td>
</tr>
</tbody>
</table>
Table 2: Comparison on compression ratio.

<table>
<thead>
<tr>
<th>CORPUS</th>
<th>etdc</th>
<th>v2vdc</th>
<th>v2vdc_H</th>
<th>Re-pair</th>
<th>ppmidi</th>
<th>gzip</th>
<th>p7zip</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALGARY</td>
<td>2.081</td>
<td>37.40%</td>
<td>35.43%</td>
<td>35.21%</td>
<td>31.20%</td>
<td>26.39%</td>
<td>36.95%</td>
<td>29.97%</td>
</tr>
<tr>
<td>FT91</td>
<td>14.404</td>
<td>35.53%</td>
<td>27.15%</td>
<td>26.65%</td>
<td>24.00%</td>
<td>25.30%</td>
<td>36.42%</td>
<td>25.53%</td>
</tr>
<tr>
<td>CR</td>
<td>49.888</td>
<td>31.94%</td>
<td>21.55%</td>
<td>23.13%</td>
<td>20.16%</td>
<td>22.42%</td>
<td>33.29%</td>
<td>21.64%</td>
</tr>
<tr>
<td>ZIFF</td>
<td>180.879</td>
<td>33.77%</td>
<td>24.01%</td>
<td>23.60%</td>
<td>20.32%</td>
<td>23.04%</td>
<td>33.06%</td>
<td>22.90%</td>
</tr>
</tbody>
</table>

Table 3: Comparison of compression and decompression time.

<table>
<thead>
<tr>
<th>CORPUS</th>
<th>etdc</th>
<th>v2vdc</th>
<th>v2vdc_H</th>
<th>Re-pair</th>
<th>ppmidi</th>
<th>gzip</th>
<th>p7zip</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALGARY</td>
<td>0.128</td>
<td>0.595</td>
<td>0.643</td>
<td>1.910</td>
<td>0.780</td>
<td>0.287</td>
<td>1.610</td>
<td>0.366</td>
</tr>
<tr>
<td>FT91</td>
<td>0.652</td>
<td>3.765</td>
<td>6.500</td>
<td>15.554</td>
<td>5.602</td>
<td>1.588</td>
<td>17.932</td>
<td>2.476</td>
</tr>
<tr>
<td>CR</td>
<td>2.054</td>
<td>15.425</td>
<td>42.960</td>
<td>69.972</td>
<td>14.441</td>
<td>5.388</td>
<td>65.002</td>
<td>9.550</td>
</tr>
<tr>
<td>ZIFF</td>
<td>7.982</td>
<td>76.250</td>
<td>558.970</td>
<td>504.230</td>
<td>55.080</td>
<td>20.667</td>
<td>248.472</td>
<td>34.887</td>
</tr>
</tbody>
</table>

On the other hand, as very long phrases typically occur just a few times, using small minFreq values enables us to choose them. However, this leads to including also low-frequent short phrases that might improve compression very slightly (in v2vdc_H) or even worsen it if no heuristic is used (v2vdc). The gain-based heuristic only partially overcomes such a problem. Although it includes only phrases that produce a gain, (a) it does not handle the combinatorial problem of not-so-good phrases preventing good phrases to be chosen if they overlap in the text, (b) it does not handle the problem that choosing phrases decreases the frequency of its words, thus flattening the histogram and hampering their zero-order compression, (c) it only estimates the final codeword length of the words and phrases.

Compression ratio. Table 2 shows the compression ratio obtained by the compressors tested. We set minFreq = 10 for our compressors\(^2\). The variants of v2vdc obtain good compression ratios when the corpus size is large enough. Our compressors improve the results of the word-based etdc by around 8-10 percentage points, and those of gzip by at more than 10 (except in the smallest files). V2vdc variants are overcome by up to 4 percentage points by re-pair. The latter benefits from using bit-oriented coding instead of dense coding: By using re-pair coupled with a dense coder, the gap decreases to around 1.5 percentage points. The non-searchable ppmidi and p7zip overcome v2vdc and v2vdc_H by around 1-2 percentage points (in the larger corpora), and bzip2 is overcome by around 2 percentage points.

Compression and decompression speed. Table 3 shows compression and decompression times. In compression v2vdc pays much time for building the suffix

\(^2\)Decompression runs around 5-10% faster for minFreq = 10 than for minFreq = 5.
array and the LCP structures\textsuperscript{3}, and \textit{v2vdc}_H has also to deal with the computation of the heuristic\textsuperscript{4} used to select good candidate phrases.

The faster compressors overcome our \textit{v2vdc} variants by far: \textit{etdc} is 5-10 times faster than \textit{v2vdc}, whereas \textit{gzip} and \textit{bzip2} are around 2-4 and 1-2 times faster than \textit{v2vdc}, respectively. \textit{V2vdc} is on a par with \textit{ppmdii} in most texts, with the exception of ZIFF corpus. The comparison among the best compressors (in compression ratio) shows that \textit{v2vdc} is from 2 to 6 times faster than \textit{re-pair} and \textit{p7zip} (which uses 2 CPUs in our test machine for both compression and decompression). Finally, we show that \textit{v2vdc}_H is typically faster than \textit{re-pair} and \textit{p7zip} (except in ZIFF corpus), and slower than the others.

With regard to decompression, \textit{v2vdc} is among the fastest decompressors. It benefits from a better compression ratio and from using a fast decompression algorithm (similar to that of \textit{etdc}), and is able to be on par with two well-known fast decompressors such as \textit{etdc} and \textit{gzip}. The only exception is the small CALGARY corpus, as most of the decompression time is devoted to recovering the compressed header in \textit{v2vdc}. The other variant, \textit{v2vdc}_H, is still more successful due to its better compression ratio and by the fact that it deals with a smaller number of phrases (the decoding loop is broken by the first occurrence of each phrase).

\textbf{Search time comparison.} We searched for single-word patterns chosen at random from the vocabulary of corpus ZIFF, following the model \cite{15} where each vocabulary word is sought with uniform probability. Those patterns were classified into three ranges of frequency: low, average, and high. Table 4 shows the average pattern length for each range and the time measures. We considered two scenarios: On the one hand we considered searches performed over plain text using our own implementation of \textit{Horspool} algorithm \cite{16} (\textit{horspool}). On the other hand, we performed searches over text compressed with \textit{etdc} using the adapted \textit{Horspool} searcher at \url{http://rosalia.dc.fi.udc.es/codes}. These are compared with our searchers over text compressed with \textit{v2vdc} and \textit{v2vdc}_H.

In the case of compressed searches, we are measuring the time needed for scanning the compressed data. We are neglecting the time needed to load the header from the compressed file, as this is paid only once and possibly amortized over many searches. The loading time is 280, 232, and 90msec. respectively, for \textit{v2vdc}, \textit{v2vdc}_H, and \textit{etdc}.

As expected \cite{4}, searches over compressed text are faster than those performed over plain text. The only exception is that \textit{horspool} on plain text overcomes \textit{v2vdc} in searches for very frequent patterns. In this scenario \textit{v2vdc} variants have to search in parallel for many phrases that will contain such word. For low and average frequency words, \textit{v2vdc} variants are able to improve the results not only of plain text searchers, but also those of \textit{etdc}. This is of particular interest as \textit{etdc} is known to be the fastest word-based technique when searching compressed text \cite{4}.

\textsuperscript{3} We used \texttt{qsort} to build the suffix array, and a simple brute-force approach for LCP. More sophisticated algorithms \cite{13, 10} could be used to speed up such tasks.

\textsuperscript{4} We must keep the vocabulary sorted by frequency to compute the size of the codeword of any phrase or word. The compression speed can be improved by using ideas similar to previous work \cite{5}.
Table 4: Comparison in search time (in milliseconds) for ZIFF corpus.

<table>
<thead>
<tr>
<th>Freq. Range</th>
<th>avg. length</th>
<th>horspool</th>
<th>etdc</th>
<th>v2vdc</th>
<th>v2vdcH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – 10^1</td>
<td>8.06 bytes</td>
<td>155.330</td>
<td>98.246</td>
<td>87.897</td>
<td>88.874</td>
</tr>
<tr>
<td>10 – 10^3</td>
<td>7.82 bytes</td>
<td>209.493</td>
<td>122.848</td>
<td>118.997</td>
<td>97.103</td>
</tr>
<tr>
<td>10^3 – 10^5</td>
<td>7.61 bytes</td>
<td>174.251</td>
<td>155.038</td>
<td>235.858</td>
<td>214.647</td>
</tr>
</tbody>
</table>

4 Conclusions

We have presented a new word-based variable-to-variable compressor called v2vdc that uses the suffix-array longest common prefix data to choose long and frequent phrases. Those phrases can be later included in the vocabulary of our compressor so that both phrases and single words will be statistically encoded (using etdc in our case). The two variants presented (v2vdc and v2vdcH) differ in the heuristic used to determine if a phrase deserves to be used.

Our technique stands among the best state-of-art text compressors. It obtains good compression ratios, which overcome bzip2 slightly, and are close to those of ppmdi, p7zip and re-pair. Our faster variant v2vdc, is slower than bzip2 at compression and obtains times similar to ppmdi, whereas it overcomes by far the time performance of re-pair and p7zip. At decompression, no other technique can compete with our v2vdc variants, which are typically twice as fast as p7zip and re-pair, 3 times faster than bzip2, and 20-30 times faster than ppmdi.

When compared with the fastest (and less powerful) compression techniques, v2vdc variants overcome gzip by around 10 percentage points, and the most successful semstatic word-based text compressors (the optimal technique, Plain Huffman [15], compresses only 3% more than etdc) by 7-10 points. As expected, we are much slower at compression than etdc and around 3-4 times slower than gzip. However, things change at decompression, where we are on a pair with the fast etdc technique.

Search capabilities are another interesting feature of the v2vdc variants. They are not only searchable but also able to overcome (except for very frequent patterns) the speed of etdc, which is the fastest searchable compressed text format.

To sum up, v2vdc raises as a new text compressor with excellent compression ratio, affordable compression speed, very fast decompression, and the ability to efficiently perform compressed text searches. In addition, random decompression is supported.

As future work, we are targeting at studying new heuristics that could lead to a better selection of phrases. Optimizing the computation of the statistics needed for such heuristics would be also of interest. Finally, building the LCP on disk is an open research problem [9] that would allow us to compress larger corpora efficiently.

References


