# Compression of a Dictionary

Jan Lánský and Michal Žemlička

Charles University, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
zizelevak@gmail.com, michal.zemlicka@mff.cuni.cz

**Abstract.** Some text compression methods take advantage from using more complex compression units than characters. The synchronization between coder and decoder then can be done by transferring the unit dictionary together with the compressed message. We propose to use a dictionary compression method based on a proper ordering of nodes of the tree-organized dictionary. This reordering allows achieving of better compression ratio. The proposed dictionary compression method has been tested to compress dictionaries for word- and syllable-based compression methods. It seems to be effective for compressing dictionaries of syllables, and promising for larger dictionaries of words.

## 1 Introduction

Dictionary is used in many applications. Sometimes the space occupied by a dictionary is important and should be taken into account. Then it is reasonable to consider storing the dictionary in a compressed form. We propose here a method for the compression of dictionaries. We have focused on dictionaries used for text compression – or even more precisely: on a compression of dictionaries used by word- or syllable-based document compression methods. The comparisons with other methods are therefore oriented to this topic.

The paper is structured as follows: At first (in part two) we describe the dictionaries and give some formal definitions. Then, in part three, we remember some existing methods used to store the dictionaries. Part four is dedicated to the newly proposed methods. The comparisons of the tested methods are presented in part five. Last part (sixth) is dedicated to the summary.

## 2 Dictionary

We suppose that a dictionary is a set of ordered pairs (*string*, *number*), where the *string* is a string over an alphabet $\Sigma$ and the *number* is an integer of the range 1–$n$ where $n$ is the number of the ordered pairs in the dictionary.

It is sometimes useful to partition the set of all *strings* (dictionary items) into several disjoint categories. It is possible that the join of the categories does not cover the set of all possible strings over $\Sigma$. In this case it is necessary to ensure that the input strings always fit in the given categories.

For the text compression purposes this requirement can be met e.g. by a proper input string selection (partition of the input message into properly formed subparts). Words and syllables are special types of such strings.

## 3 Existing Methods for the Compression of a Dictionary

It is quite common for the papers on word- and syllable-based compression methods that their authors give no big importance to the compression of the dictionary as the dictionary often makes only a small part of the resulting compressed message. It is probably true for very large documents but for middle-sized documents the importance of a dictionary size grows as the dictionary takes larger part of the compressed message.

The following two approaches are the most widely used: The first approach is based on coding of a succession of strings (words or syllables) contained in it. In the second approach the dictionary is compressed as a whole. All the strings are concatenated using special separators. The resulting file is then compressed using some general method.

### 3.1 Compression of Dictionary character-by-character – CD

There is described a method in [1] for the encoding of strings using a partitioning of the strings into five categories, similarly to the method TD3 described below. Every string is encoded as a succession of string type codes followed by encoded string length and by the codes for individual symbols. String type is encoded using binary phase coding (*c1*), string length is encoded by adaptive Huffman code (*c2*), and individual symbols are coded also using adaptive Huffman code (letters by $c3$, numbers by $c4$, and other characters by $c5$). Lower and upper case letters use the same code value $c3$, they are distinguished by the syllable type. All adaptive Huffman trees are initialized according language specification. Examples are given in Fig. 1.

```
code("to") = c1(mixed), c2(2), c3('t'), c3('o')
code("153") = c1(numeric), c2(3), c4('1'), c4('5'), c4('3')
code(". ") = c1(other), c2(2), c5('.'), c5('0')
```

**Fig. 1.** An example of a coding a string by the CD method

It is not necessary to know the whole dictionary at the beginning. It is possible to compress individual items on the fly. It is then possible to encode new items whenever they are encountered. Other methods discussed in this paper need to compress the whole dictionary at once.

### 3.2 External Compression of a Dictionary

Let us have a separator being not part of the used alphabet. Let all the strings forming the dictionary are concatenated to a single string using this separator. The resulting string is then encoded using an arbitrary compression method. In [2] the authors tried to encode the dictionary of word using gzip, PPM, and bzip2 methods and recognized as best for this purpose bzip2. We tried to encode the dictionary using bzip2 [3] (in the tables denoted as BzipD – bzip compressed dictionary) and LZW [4] (denoted in the tables as LZWD – LZW compressed dictionary).

## 4 Trie-Based Compression of a Dictionary

When designing here introduced methods TD1, TD2, and TD3 we decided to represent the dictionary by a data structure *trie* [5, Section 6.3: Digital Searching, pp. 492–512]. Trie $T$ is a tree of maximal degree $n$, where $n$ is the size of the alphabet of symbols $\Sigma$ and satisfies following conditions: The root represents an empty element. Let the string $\alpha$ be represented by the node $A$, the string $\beta$ represented by the node $B$. If the node $A$ is father of the node $B$, then the string $\beta$ is created by concatenation of the string $\alpha$ by one symbol from $\Sigma$. For all nodes $A$ and $B$ exists a node $C$, that represents common prefix of strings $\alpha$ and $\beta$ and this node is on both paths (including border points) from the root to $B$ and from the root to $A$.

The dictionary trie is created from the strings appearing in the text. Then the trie is encoded. Duriung this encoding there is a unique number assigned to each string using depth-first traversal of the trie.

### 4.1 Basic Version – TD1

Trie compression of a dictionary (TD) is based on coding structure of a trie representing the dictionary. For each node in the trie we know the following: whether the node represents a string (*represents*), the number of sons (*count*), the array of sons (*son*), and the first symbol of an extension for each son (*extension*). Basic version of such encoding (TD1) is given by a recursive procedure *EncodeNode1* in Fig. 2 which traverse the trie by a depth first search (DFS) method. For encoding the whole dictionary we call this procedure on the root of the trie representing the dictionary.

In procedure *EncodeNode1* we code only a number of sons and the distances between the extensions of sons. For non-leaf nodes we must encode in one bit whether that node represents a dictionary item (e.g. syllable or word) or not. Leafs represent dictionary items always, it is not necessary to code it. Differences between extensions of the sons are given as distances of binary values of the extending characters. For coding of a number of sons and the distances between them we use gamma and delta Elias codes [6]. We have tested other Elias codes too, but we achieved the best results for the gamma and delta codes. The numbers of sons and the distances between them can reach the value 0, but standard

```
EncodeNode1(node) {
  output->WriteGamma0(count + 1); /* We encode number of sons */
  if (count = 0) return;
  /* Mark whether the node represents a string*/
  if (represents)
    output->WriteBit(1);
    else output->WriteBit(0);
  previous = 0;
  /* We iterate and encode all sons of this modes */
  for(i = 0; i < count; i++) {
     /* We count and encode distance between sons */
    distance = son[i]->extension - previous;
    output->WriteDelta0(distance + 1);
    /* Recursive calling of procedure on the given son */
    EncodeNode1(son[i]);
    previous = son[i]->extension;
  }
}
```

**Fig. 2.** Procedure EncodeNode1

versions of gamma and delta codes starts from 1 what means that these codings do not support this value. We therefore use slight modifications of Elias *gamma* and *delta* codes: $gamma_0(x) = gamma(x + 1)$ and $delta_0(x) = delta(x + 1)$.

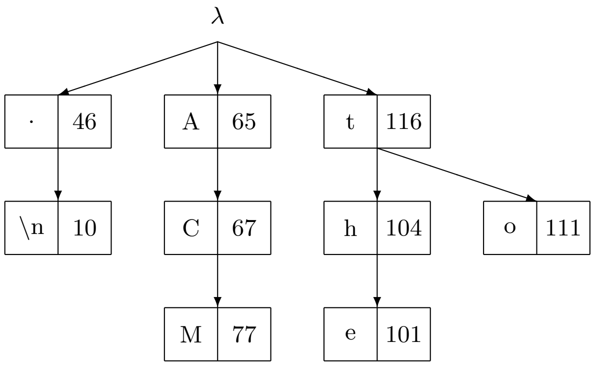

**Fig. 3.** Example of dictionary for TD1

An example is given in Fig. 3. The example dictionary contains the strings `".\n"`, `"ACM"`, `"AC"`, `"to"`, and `"the"`. Let us introduce the TD1 method by coding the root of the trie representing our example dictionary:

In the node we must first encode the number of its sons. Root has 3 sons, hence we say that $gamma_0$-code of the 3 (sons) is a string of bits '00001' and we write $gamma_0(3) = 00001$.

Then we state that the already represented word (an empty string) is not part of the dictionary by writing a bit 0.

Value of the the first son is encoded as a distance between its value and zero by $delta_0(46 - 0) = 0100101111$.

Then the first subtrie is encoded by a recursive call of the encoding procedure on the first son of the actual node.

When the first subtrie is fully encoded, we should specify what the second son is. The difference between the first and the second son is $65 - 46$, hence we write $delta_0(65 - 46) = 000110011$.

Then we encode the second subtrie and the third son and the subtrie rooted in it. Now the whole node and all it subtries are encoded. As our example node is the root, we have encoded the whole trie representing the dictionary.

## 4.2 Version with Translator – TD2

In TD1 version the distances between sons according binary values of the extending symbols are coded. These distances are encoded by Elias delta coding representing smaller numbers by shorter codes and larger numbers by longer codes. In version TD2 we reorder the symbols in the alphabet according the types of the symbols and their frequencies typical for given language. In our exmaple the symbols 0–27 are reserved for lower-case letters, 28–53 for upper-case letters, 54–63 for digits and 64–255 for other symbols. There are some examples in table 1.

**Table 1.** An example of new ordering of the symbols

| symbol | 'e' | 't' | 'a' | 'I' | 'T' | 'A' | '0' | '1' | '2' | ' ' | ',' | '.' |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ord(symbol) | 0 | 1 | 2 | 28 | 29 | 30 | 54 | 55 | 56 | 64 | 65 | 66 |

Improving procedure TD1 by a replacement of the expression "$son[i] \rightarrow extension$" by the expression "$ord(son[i] \rightarrow extension)$" in the lines 08 and 11 we get procedure TD2 (Fig. 4).

Let us demonstrate this method on an example (Fig. 5). The example dictionary contains again the strings `".\n"`, `"ACM"`, `"AC"`, `"to"` and `"the"`. We will describe the work of the coding procedure EncodeNode2 on the node labelled by 't'.

In a node we must first encode the number of its sons. Our node has two sons, hence we write $gamma_0(2) = 011$.

Then we state that the already represented word (the string "t") is not part of the dictionary by writing a bit 0.

```
EncodeNode2(node) {
  output->WriteGamma0(count + 1); /* We encode number of sons */
  if (count = 0) return;
  /* Mark whether the node represents a string*/
  if (represents)
    output->WriteBit(1);
    else output->WriteBit(0);
  previous = 0;
  /* We iterate and encode all sons of this modes */
  for(i = 0; i < count; i++) {
     /* We count and encode distance between sons */
    distance = ord(son[i]->extension) - previous;
    output->WriteDelta0(distance + 1);
    /* Recursive calling of procedure on the given son */
    EncodeNode2(son[i]);
    previous = ord(son[i]->extension);
  }
}
```

**Fig. 4.** Procedure EncodeNode2

Value of the the first son of `'t'` is encoded as a distance between its value 3 and zero by $delta_0(3 - 0) = 01100$.

Then the first subtrie of node `'t'` is encoded by a recursive call of the encoding procedure on the first son of the actual node.

When the subtrie of node `'t'` is fully encoded, we should specify what the second son of the root is. The difference between first and second son is $6 - 3$, hence we write $delta_0(6 - 3) = 01100$.

Then we encode second subtrie. Now the whole node and all it subtries are encoded.

## 4.3 Version Using Types of Strings – TD3

Words and syllables are special types of strings. We recognize these five types of words (and syllables): lower-words (from lower-case letters), upper-words (from upper-case letters), mixed-words (having the first letter upper-case and the following letters lower-case), numeric-words (from digits) a other-words (from special characters). We know the type of a coded string for some nodes in the trie (in Fig. 6 *IsKnownTypeOfSons*) and we can use this information.

If a string begins with a lower-case letter (lower-word or lower-syllable), the following letters must be lower-case too. In a trie each son of a lower-case letter can be only a lower-case letter too. Similar situation is for other-words and numeric-words. If a string begins with an upper-case letter, we must look at the second symbol to recognize the type of the string (mixed or upper). In our example (Fig. 5) we know for the nodes `'t'`, `'o'`, `'h'` and `'e'` that all their sons are lower-case letters.

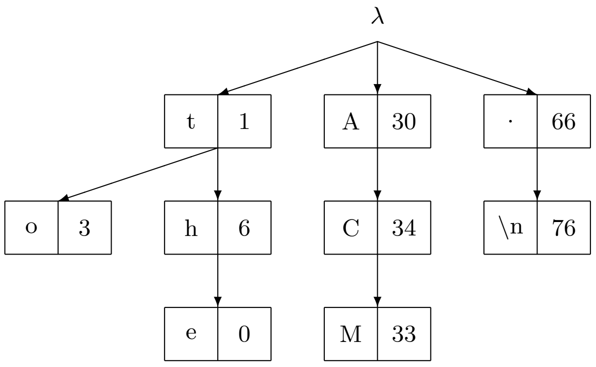

**Fig. 5.** Example of a dictionary for TD2 and TD3

In the new ordering described in version TD2 it is given for each symbol type some interval of the new orders. Function first returns for each type of symbols the lowest orders available for given symbol type. Function *first* is described in Tab. 2.

**Table 2.** Values of function first

| type of symbols | lower-case letter | upper-case letter | digit | other |
|---|---|---|---|---|
| first(type) | 0 | 28 | 54 | 64 |

We are counting (Fig. 6, line 10) and coding (Fig. 6, line 11) the distances between the sons. For the first sons of some nodes of a known type, we can use function *first* and decrease the value of the distance and shorten the code. We modify version TD2 by a modifying of the line 06 and inserting the lines 07 and 08 getting version TD3.

Let us show the differences between TD3 and TD2 on our example (Fig. 5).

Let us go directly to the node `'t'`. Here we must first encode the number of the sons of this node (2), we write $gamma_0(2) = 011$.

Then we state that the already represented word (string `"t"`) is not part of the dictionary by writing a bit 0.

Value of the the first son of our node is encoded as a distance between its value (3) and and zero (it is the first son) decreased by value of function $first$ for a lower-case letter (0). Encoded value is $delta_0(3-0-0) = 01100$. It is possible to restrict the shift interval by $first$ of the encoded character type as we know this type – in a subtrie of the node `'t'` occur only lower-case letters. The encoded value is the same as in TD2 but there is a diference is in the calculation.

Other codings are made accordingly.

```
EncodeNode2(node) {
  output->WriteGamma0(count + 1); /* We encode number of sons */
  if (count = 0) return;
  /* Mark whether the node represents a string*/
  if (represents)
    output->WriteBit(1);
    else output->WriteBit(0);
  /* Using knowledge of type of node (improvent of method TD3) */
  if (IsKnownTypeOfSons)
    previous = first(TypeOfSymbol(This->Symbol))
    else previous = 0;
  /* We iterate and encode all sons of this node */
  for(i = 0; i < count; i++) {
    /* We count and encode distances between sons */
    distance = ord(son[i]->extension) - previous;
    output->WriteDelta0(distance + 1);
    /* Recursive calling of the procedure on the given son */
    EncodeNode2(son[i]);
    previous = ord(son[i]->extension);
  }
}
```

**Fig. 6.** Procedure EncodeNode3

## 5 Results

We have tested three versions of the method compressing the dictionary using the trie data structure (TD – variants TD1, TD2, TD3), one method compressing the dictionary character-by-character (CD), and two methods using an external compressing tool for the concatenated directory items (LZWD, BzipD).

We have tested the dictionaries of words and syllables for variously sized documents written in following three languages: English (EN), German (GE), and Czech (CZ).

The best for the dictionaries of syllables it appears to be the method TD3 that outperfomed all other tested methods on all tested document sizes. For example, when compressing a 10KB document, TD3-compressed dictionary takes about 770 bytes whereas the second best method (CD) takes about 1450 bytes. In the case of the compression of dictionaries of words the best-performing method has been for small documents (up to 10kB) CD, for middle-sized documents BzipD, and for large documents TD3. The boundary between 'middle-sized' and 'large' documents is in this case dependent on the used language: for Czech it was about 50kB, for English about 200kB and for German about 2MB.

It seems that the success of the TD methods (TD3 inclusive) grows with the average arity of the trie nodes. The syllables are short and the trie representing a dictionary of syllables is typically dense, hence the TD3 method has been always the best.

**Table 3.** Dictionary of syllables: Compression ratio (Compared with the size of a whole file) in bits per character

| — | File size | 100 B | 1 kB | 10 kB | 50 kB | 200 kB | 500 kB | 2 MB |
|---|---|---|---|---|---|---|---|---|
| Lang. | Method | 1 kB | 10 kB | 50 kB | 200 kB | 500 kB | 2MB | 5 MB |
| CZ | LZWD | 5.359 | 3.233 | 1.423 | 0.562 | 0.343 | 0.204 | — |
| CZ | CD | 3.741 | 2.432 | 1.130 | 0.461 | 0.284 | 0.169 | — |
| CZ | BzipD | 5.285 | 2.952 | 1.227 | 0.468 | 0.285 | 0.168 | — |
| CZ | TD1 | 4.124 | 2.232 | 0.870 | 0.315 | 0.185 | 0.115 | — |
| CZ | TD2 | 2.944 | 1.594 | 0.638 | 0.240 | 0.143 | 0.093 | — |
| CZ | TD3 | **2.801** | **1.532** | **0.612** | **0.226** | **0.134** | **0.081** | — |
| EN | LZWD | 4.580 | 1.715 | 0.732 | 0.426 | 0.269 | 0.152 | 0.059 |
| EN | CD | 2.983 | 1.287 | 0.583 | 0.360 | 0.234 | 0.133 | 0.052 |
| EN | BzipD | 4.390 | 1.523 | 0.626 | 0.353 | 0.222 | 0.124 | 0.047 |
| EN | TD1 | 3.792 | 1.276 | 0.506 | 0.272 | 0.158 | 0.086 | 0.033 |
| EN | TD2 | 2.871 | 0.954 | 0.384 | 0.212 | 0.124 | 0.069 | 0.028 |
| EN | TD3 | **2.666** | **0.890** | **0.354** | **0.195** | **0.116** | **0.063** | **0.024** |
| GE | LZWD | 4.259 | 2.995 | 1.139 | 0.580 | 0.345 | 0.202 | 0.104 |
| GE | CD | 3.068 | 2.360 | 0.997 | 0.530 | 0.315 | 0.185 | 0.091 |
| GE | BzipD | 4.127 | 2.689 | 0.949 | 0.479 | 0.285 | 0.166 | 0.087 |
| GE | TD1 | 3.952 | 2.539 | 0.832 | 0.377 | 0.207 | 0.122 | 0.045 |
| GE | TD2 | 3.020 | 1.914 | 0.627 | 0.284 | 0.157 | 0.097 | 0.035 |
| GE | TD3 | **2.730** | **1.805** | **0.599** | **0.275** | **0.150** | **0.086** | **0.033** |

German language has a lot of different and long word forms, the trie representing such dictionary is quite sparse and therefore the TD3 method outperformed other methods only for dictionary of very large documents.

English typically uses less word forms than Czech and German. These word forms are often shorter than the ones used in Czech and German. The trie is then for smaller documents quite sparse and therefore our compression method outperforms the other ones only for larger documents.

In Czech the documents are typically made form lots of middle-sized word forms and the dictionary tries are therefore quite dense. It is the reason why the method has been so successful for the dictionaries of Czech documents.

## 6 Conclusion

We have proposed three methods for compression of dictionaries based on the representation of the dictionary by a trie data structure. One of them (TD3) has compressed the dictionary of syllables for given files better than all other tested methods have. It has been also the most successful method for compression of dictionaries of words of large documents.

Such dictionaries are used by many word- and syllable-based compression algorithms. Improving compression ratio of the dictionary improves (although with smaller impact) the overall compression ratio of these methods.

**Table 4.** Dictionary of words: Compression ratio (Compared with the size of a whole file) in bits per character

| — | File size | 100 B | 1 kB | 10 kB | 50 kB | 200 kB | 500 kB | 2 MB |
|---|---|---|---|---|---|---|---|---|
| Lang. | Method | 1 kB | 10 kB | 50 kB | 200 kB | 500 kB | 2MB | 5 MB |
| CZ | LZWD | 5.984 | 4.549 | 3.076 | 1.934 | 1.557 | 1.161 | — |
| CZ | CD | **4.378** | **3.830** | 2.948 | 1.968 | 1.648 | 1.260 | — |
| CZ | BzipD | 5.784 | 4.045 | **2.559** | 1.582 | 1.255 | 0.921 | — |
| CZ | TD1 | 8.443 | 6.520 | 4.146 | 2.250 | 1.713 | 1.178 | — |
| CZ | TD2 | 5.935 | 4.531 | 2.874 | 1.550 | 1.176 | 0.814 | — |
| CZ | TD3 | 5.781 | 4.462 | 2.844 | **1.534** | **1.167** | **0.800** | — |
| EN | LZWD | 4.699 | 2.195 | 1.203 | 0.872 | 0.687 | 0.443 | 0.189 |
| EN | CD | **3.100** | **1.776** | 1.095 | 0.847 | 0.695 | 0.454 | 0.197 |
| EN | BzipD | 4.508 | 1.915 | **1.002** | **0.714** | 0.563 | 0.361 | 0.154 |
| EN | TD1 | 6.320 | 3.144 | 1.698 | 1.108 | 0.813 | 0.498 | 0.191 |
| EN | TD2 | 4.526 | 2.142 | 1.144 | 0.753 | 0.554 | 0.341 | 0.132 |
| EN | TD3 | 4.219 | 2.062 | 1.110 | 0.734 | **0.544** | **0.333** | **0.128** |
| GE | LZWD | 4.712 | 3.634 | 1.819 | 1.227 | 0.996 | 0.706 | 0.716 |
| GE | CD | **3.582** | **3.091** | 1.787 | 1.293 | 1.096 | 0.799 | 0.789 |
| GE | BzipD | 4.409 | 3.216 | **1.506** | **1.001** | **0.797** | **0.558** | 0.565 |
| GE | TD1 | 7.187 | 5.748 | 2.585 | 1.700 | 1.383 | 0.945 | 0.844 |
| GE | TD2 | 4.985 | 3.885 | 1.691 | 1.094 | 0.875 | 0.601 | 0.534 |
| GE | TD3 | 4.699 | 3.776 | 1.660 | 1.085 | 0.867 | 0.591 | **0.532** |

## References

1. Lánský, J., Žemlička, M.: Text compression: Syllables. In Richta, K., Snášel, V., Pokorný, J., eds.: DATESO 2005, Prague, Czech Technical University (2005) 32–45 Available from http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS//Vol-129/paper4.pdf.
2. Isal, R.Y.K., Moffat, A.: Parsing strategies for BWT compression. In: Data Compression Conference, Los Alamitos, CA, USA, IEEE CS Press (2001) 429–438
3. Seward, J.: (The bzip2 and libbzip2 official home page) http://sources.redhat.com/bzip2/ as visited on 6th February 2005.
4. Welch, T.A.: A technique for high performance data compression. IEEE Computer **17** (1984) 8–19
5. Knuth, D.: The Art of Computer Programming, Volume 3: Sorting and Searching. Third edn. Addison-Wesley (1997)
6. Elias, P.: Universal codeword sets and representation of the integers. IEEE Trans. on Information Theory **21** (1975) 194–203