

Low-Complexity Compression of Short Messages

Stephan Rein, Clemens Gühmann, Frank H.P. Fitzek *

Technical University of Berlin,

Dept. of Electronic Measurement and Diagnostic Technology

* Aalborg University, Dept. of Telecommunication Technology

[stephan.rein|clemens.guehmann]@tu-berlin.de ff@kom.aau.dk

Abstract

We describe a low-complexity scheme for lossless compression of short text messages. The method uses arithmetic coding and a specific statistical context model for prediction of single symbols. Our particular contribution is a simple yet effective approach for storing highly complex statistics in a succinct yet effective data model that can easily be trained by text data. The proposed model already gives good compression rates with a RAM memory size of 128 kByte, thus making lossless data compression with statistical context modeling readily applicable to small devices like wireless sensors or mobile phones.

1: Introduction and Motivation

Compression of short messages is a vital operation for low complexity entities such as mobile phones or wireless sensors. A wireless sensor is characterized by a much larger power consumption in information transmission over the wireless medium than on the operational part. In the mobile phone's world the compression would allow users to use more SMS characters than the provider's default value. Therefore, compression of short messages to be conveyed over the wireless medium seems to be promising in terms of power, costs, and bandwidth savings.

Prediction by partial matching (PPM) is a lossless data compression scheme, where a single symbol is coded taking its previous symbols into account, which are called the symbol's context. A context model is employed that gives statistical information on a symbol and its context. The encoder uses specific symbols to signal the decoder the current context. The number of context symbols defines the model order and is a basic parameter for the compression rate and the algorithm complexity. The symbol probabilities can be processed by an arithmetic coder, thus achieving superior compression over many widespread compression schemes, as for instance the *Ziv-Lempel* methods (LZ77,LZ78). However, PPM is computationally more complex and even its recent further developments do not allow its application to small, battery - driven devices.

In this paper, we address two general problems with PPM, a) the large RAM memory requirements and b) the typically poor compression of PPM for the first sequence

of symbols. We solve a) with a single, specific hash table array that combined with a hash key and a parity check form a data model for complex statistics. We address b) by usage of a static context model with an elementary dimension reduction technique for efficient storage of the model in the RAM memory. The model is trained with a text file that is in ideal case typical for the data to be compressed. We employ a computationally less demanding static context model in contrast to the general concept of adaptive context modeling in PPM, where the statistics are gathered and updated during the compression.

This paper is organized as follows: In the next subsection, we review related literature. In section 2, we describe the model we use for computing the statistics and detail how it is trained using a dimension reduction technique. We further give an elementary method for compression of the statistical context model. Thus, a battery driven device can maintain a set of statistical models for different types of data. In section 3, we analyze the statistical accuracy of our data model, which depends on the ability of the hash function to foresee the data sequences to be compressed. In section 4, we give compression results for text files from four different corpora. Finally, we summarize our findings in section 5.

1.1: Related Literature

There exists a large body of literature on lossless data compression. Most of the work concerns the improvement of compression performance while reducing the computational requirements, see for instance [4, 5, 13, 9]. In [6], an optimal statistical model (SAMC) is adaptively constructed from a short text message and transmitted to the decoder. Thus, such an approach is not useful for short message compression, as the overall compression ratio would suffer from the additional size of the context model. In [12], the scheme PPM with *information inheritance* is described, which improves the efficiency of PPM in that it gives compression rates of 2.7 bits/byte with a memory size of 0.6 MBytes, which makes PPM readily applicable to low-cost computers. The memory requirements for ZIP, BZIP2, PPMZ, and PPMD were shown to vary from 0.5 to more than 100 MByte.

The most related work that we are aware of is the recent study in [8], where a tree machine is employed as a static context model. It is shown that zip (Info-ZIP 2.3), bzip-2 (Julian Seward version 1.0.2), rar (E. Roshal, version 3.20 with PPMII), and paq6 (M. Mahoney) fail for short messages (compression starts for files larger than 1000 Bytes). In contrast to our work, the model is organized as a tree and allocates 500 kBytes of memory, which makes the proposed method less feasible for a mobile device. In addition, our method is conceptually less demanding, which is a key requirement for low-complexity devices. To our best knowledge, this is the first paper that combines lossless short message compression with a low-complexity context modeling scheme.

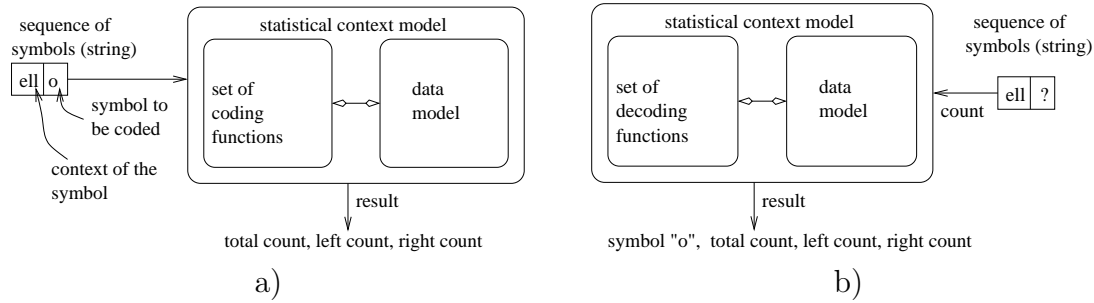


Figure 1. The functionality of a statistical context model. Fig. a) illustrates the requirements for encoding, where the appropriate statistics are returned for each string. Fig. b) illustrates the function of the model for decoding, where a symbol count is mapped to the decoded symbol. Our particular contribution is the data model, which is illustrated in Fig. 2 a).

2: Context Model

2.1: Statistical Context Model

When encoding a message, each symbol and its context (which is typically a string of bytes) are given to a statistical context model, which returns/estimates the appropriate statistics, as illustrated in Fig. 1. A widely-used method for storing the statistics is a data tree, where each node contains statistics for a given string. To access the nodes, functions for traversing the tree have to be implemented, which in general are computationally demanding. The nodes do not only contain the statistics, but also need pointers to link them.

To reduce the required memory and the complexity, we use a context model that consists of a *data model* and functions for estimating the statistics (i.e. a *total*, a *left*, and a *right count*) on the fly with standard techniques, see [2, 3] for details. The data model contains the symbol counts, where a single symbol refers to a node in a data tree. The task of the data model is to return the appropriate count to any character array. The model consists of a specific hash function, a data array each element containing two entries, and a parity check function, as illustrated in Fig. 2 a). It works as follows. First, the string is given to a *One-at-a-Time Hash* hash function [7], which transfers the string of characters to an integer number within the interval $[0, TableSize - 1]$, where *TableSize* denotes the total number of elements in the hash table. The estimated hash table element contains a count and a parity byte. Then, the parity of the queried string is computed and compared with the parity in the hash table. If both parities are the same, the data model returns the count in the hash table element. Otherwise, a zero count is returned indicating that the string is not in the model. The parity check is essential as during the encoding/decoding process a large amount of strings are queried the data model for the on the fly estimation of the symbol statistics.

We note that the lossless compression/decompression with the data model even

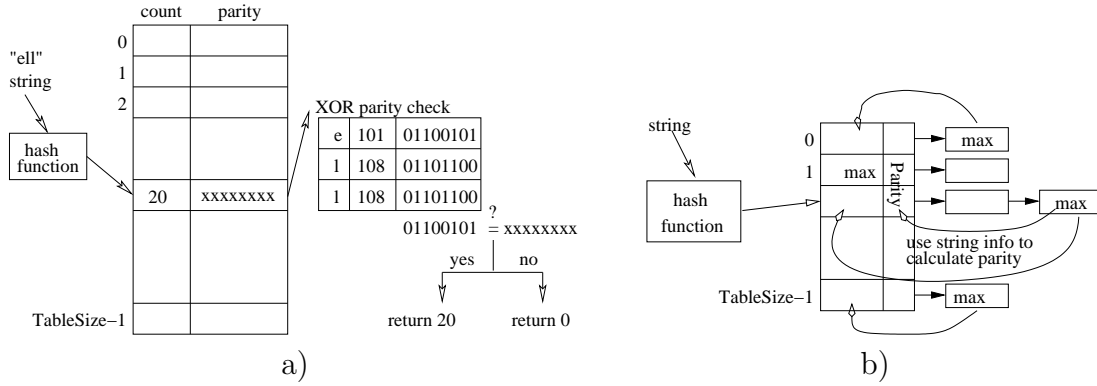


Figure 2. The data model. Fig. a) illustrates the elementary parts of the model, a hash table each element containing a count and a parity, a hash function, and a parity check function. Fig. b) illustrates the hash table of the original, more complex data model with collision resolving linked lists, which is employed to train the low-complexity model.

works without the parity check, however, the compression performance is significantly deteriorated. Let N denote the model order. For each order, the hash function maps the space of 256^{N+1} different strings to an array of $TableSize$ elements, resulting in hash function collisions which are not resolved. The task of the hash function is to only map the statistically significant context nodes.

A particular advantage of our data model is that it is parametrized by just one parameter, the size of the hash table. Each element of the hash table refers to an arbitrarily long string. Thus, the model works for each statistical context model order without any changes in the implementation. In the next section, we give a method for training the low complexity data model.

2.2: Training the model

As detailed in section 2.1, the statistical model consists of an array of elements each of them containing a count and a parity. To calculate these numbers we use a more complex statistical context model for adaptive modeling, where the data model consists of a hash table with collision resolving techniques, as illustrated in Fig. 2 b). The training of the context model is given as follows. The text file from which the training data shall be extracted of is compressed while the more complex hash table with linked elements is filled with statistical data. Each of the elements contains a string for identification of the symbol and its context. When the compression is finished, the amount of statistical data in the data model is reduced in that a) each linked list is searched for the maximum statistical item count, and b) the maximum counts are then written into the first elements of the linked list, as illustrated in Fig. 2 b). The remaining elements of the linked lists are deleted. The appending strings to the maximum counts of the table are now recalled to calculate the parities. A parity of one table element is calculated by a logical exclusive-OR operation of the

file	memory [kByte]	size [Byte]	file	memory [kByte]	size [Byte]
o2tb-32	32	16368	o3tb-32	32	16450
o2tb-64	64	30408	o3tb-64	64	32892
o2tb128	128	63060	o3tb128	128	117106
o2tb256	256	73260	o3tb256	256	176490

a)

b)

Table 1. Sizes of the compressed versions of the context model for order 2 in Fig. a) and order 3 in Figure b). A low-complexity device can maintain a set of models for different types of data.

single symbols of one string. The keys to the table elements are now discarded, thus obtaining a low complexity data model. The final result is an array of elements each element containing a maximum count and a parity. In the next section we detail a low-complexity method for efficiently storing the so obtained statistical model. Thus, a mobile device can store various types of data (for instance on its flash ROM) as it may manage many different statistical models.

2.3: Compression of the model

A mobile device does not need to permanently retain the statistical model in its RAM memory. When the compression program is not needed, the statistics can be kept in the program memory. In this section, we describe how the array of counts and parities can be efficiently stored. For our measurements, we use models for hash table lengths 16384, 32768, 65536, and 131072 elements. As each element requires 2 bytes (symbol count and parity), the statistical models allocate 32, 64, 128, and 256 KBytes of RAM memory, respectively. The models are constructed from the text file *book2* from the Calgary corpus [2].

A simple yet effective method for compression of the statistical data can be achieved due to the fact that even with a good hash, many of the elements of the hash table are empty and not employed for the statistical model. An empty element is defined by a zero count. If an element is empty, there is no need for storage of a parity. To store the counts and the parities the hash table is traversed from its first to the last element. For each element, the count and the parity is sequentially stored. In case of a zero count, a zero byte and a byte for the total number of zero elements in a stream is written on disk. Thereby, large streams of zero elements are stored by just two bytes. Typically, there exist many streams of zero elements in the data model. The maximum stream size that can be denoted is 256 empty elements. Table 1 gives the size of different compressed data models in Bytes for the orders 2 in table a) and 3 in table b). We use the so constructed models to obtain the compression results in section 4.

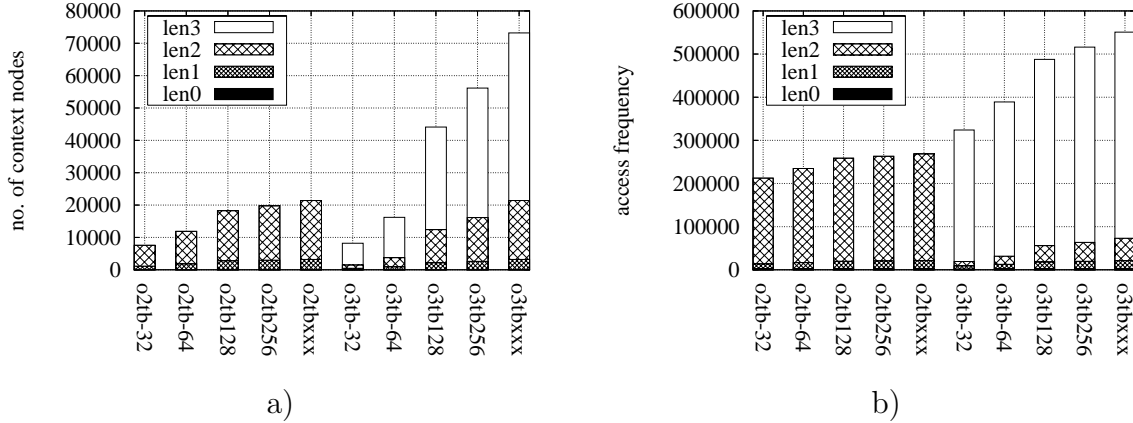


Figure 3. A data model represents the nodes of a data tree, each node referring to a different symbol with a certain context. Fig. a) classifies the nodes by their context length, Fig. b) depicts the frequency the nodes of a certain context length are accessed. The deep context nodes are more frequently accessed and should be retained by a good model.

3: Finding a good Hash

In this section we review the ability of our data model to retain the statistically significant context nodes of a data tree. Each data model consists of context nodes that refer to keys of the length $N + 1$ Bytes, where N denotes the model order. Recall that there exist 256^{N+1} different keys for each order, thus the space of keys for the higher orders is enormously larger and expectedly more difficult to model. The set \mathbf{S} of all possible inputs for the hash function is given as

$$\mathbf{S} = \sum_{order=0}^{N_{max}} 256^{N+1} \quad (1)$$

and has to be mapped to the space of $1 \dots Table$ elements, where $Table$ denotes the hash table size. The hash function should distribute the statistically important context nodes uniformly on the hash table.

In Fig. 3, we give statistics performance results for the models obtained from the training file *book2* with 32, 64, 128, and 256 kBytes of allocated RAM memory for the maximum orders 2 and 3 (denoted as described in table 1), respectively. We compare the models with the data model of *PPMc* (denoted as *tbxxx*) that is employed without memory constraints. The measurements were conducted with the *One-At-a-Time* hash key, which is detailed and evaluated in [7] to perform without collisions for mapping a dictionary of 38470 English words to a 32-bit result. Fig. 3 a) shows the number of context nodes each model contains, where *len0* to *len3* refer to the order of the key that is mapped (for instance, *len1* refers to a key of 2 bytes and the model order 1). The models o2tb-32 and o2tb-64 discard more or less half of the original model space, which is given by o2tbxxx as 21400. The models o2tb128

and o2tb256 tend to preserve the space of occurring context nodes, and therefore are expected to result into good compression performance. For order 3, the space of possible context nodes is given as 73191. The models o3tb-32 and o3tb-64 preserve less than 16200 context nodes, which is just 22% of the original model space. The models o3tb128 and o3tb256 preserve 60% and 77%, respectively.

A low-complexity data model consists of a subspace of the original data model, which should contain the context nodes that are of statistical importance. An indicator to assess the statistical quality of the retained nodes may be the frequency each node is accessed during the compression, which is illustrated in Fig. 3 b). Especially for the order 3 models, we conclude that a node of a higher order is on average more frequently accessed than a node of a lower order, for instance the access frequency for len3 context nodes is 452601 for model o3tb256, which is 88% of all node accesses. However, context nodes of order 3 just allocate 71% of the data structure.

4: Performance Evaluation

In this section we evaluate the compression performance for the low-complexity compression scheme, which we call from now *lcc* (low complexity compression). We employ English text files from the *Canterbury* corpus [1] (*alice29*, *asyoulik*, *plrabn12*), the *project Gutenberg* [10] (*hrom110*, *hrom220*), the *Calgary* [2], and the *large corpus* (*bible*, *world192*, available at <http://corpus.canterbury.ac.nz/descriptions>).

We first evaluate the compression performance of *lcc* with the training file *book2* from the Calgary Corpus and compare it with the performance of PPMc with a pre-loaded static model from *book2*. Importantly, the PPMc implementation has no memory constraints, whereas the *lcc* scheme is applied with memory sizes ranging from 32 to 256 kBytes.

Fig. 4 a) shows the compression ratios for the complete array of text files for model order 2. A remarkable difference in compression is obvious for the memory sizes 32, 64, and 128 kBytes. The compression for the Calgary Corpus and files from the Gutenberg project varies from 4.25 to 4.3 for a memory size of 64 kByte, and from 3.75 to 3.8 Bits/Byte for a memory of 128 kByte. The further duplication of memory does result into a small performance improvement of approximately 0.1 Bits/Byte. This is due to the fact that the compression for 128 kByte is already very close to the compression of the PPMc algorithm with no memory constraints, which similarly performs 0.1 Bits/Byte better than the 256 kByte *lcc* scheme.

Fig. 4 b) illustrates the compression performance for model order 3. Remarkably, *lcc* performs worse with the memory sizes of 32 and 64 kByte than for the model order 2. This is due to the fact that the models of higher order have more context nodes and overstrain the smaller hash table. In this case, the compression with 32 kBytes even does not work at all for the data file *world192*, *alice29*, and *asyoulik* and with 64 kB for the file *world192*. The 128 kByte order 3 model gives better compression than the 64 kByte order 2 model for most of the text files. The 256 kByte order 3 model achieves better compression rates than the best *lcc* order 2 model for the text files *paper1* to *paper4*. PPMc with the preloaded model performs approximately

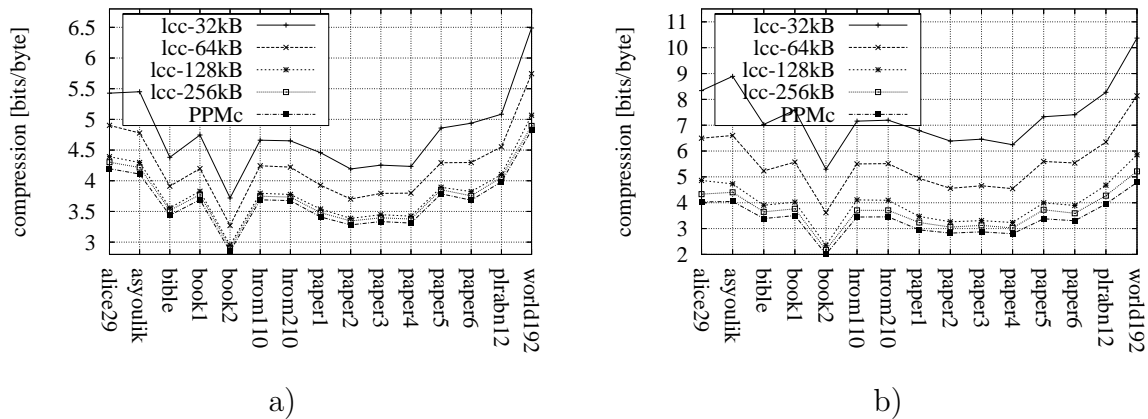


Figure 4. Compression performance for lcc with training data from *book2* for the order 2 in a) and the order 3 in b). With more memory the lcc models perform very close to the PPMc model without memory constraints.

0.2 bits per byte better than the 256 kByte order 3 lcc scheme. The 256 kByte lcc schemes for order 2 and order 3 are both very close to the PPMc compression performance.

We note that the evaluation results from Fig. 4 were not obtained to prove the compression performance of lcc for longer text files, but to indicate that the larger lcc context models retain the statistical significant data of the PPMc context model, which we employed without memory constraints. Fig. 5 shows the compression ratios for short text sequences of the file *paper4* for lcc with training data from *book2*. The data points are depicted together with the 95% confidence intervals. lcc is compared with the PPMc compression scheme using adaptive context modeling. Fig. a) shows that for model order 2 the compression rates nearly stay constant for text sequences larger than 75 Bytes, which similarly holds true for the model order 3 in Fig. b) with the exception of the 32 kByte order 3 model, which performs comparatively poor like the two PPMc adaptive context models. The order 2 models of 32, 64, and 128 kByte perform around the compression rates 4.3, 4, and 3.5 Bits/Byte, respectively. Doubling the memory size again does not result into compression improvements. The 64 and 128 KByte order 3 models perform with approximatively 4.5 and 3.4 Bits/Byte, respectively. The 256 kByte order 3 model gives the best performance of approximatively 3.1 Bits/Byte for the larger text sequences.

The results show that especially the order 2 lcc model gives reasonable compression performance while preserving low computational requirements. For higher model orders, the size of the hash table has to be enlarged.

We note that in the recent study in [8], compression ratios ranging from 2.95 to 3.1 for compression of the file *hrom210* using training data from *book1* is achieved. The better compression performance is obtained with the cost of higher computational complexity and memory requirements of 512 kBytes. In contrast to this approach, lcc is designed as a good and scalable trade-off between complexity and compression

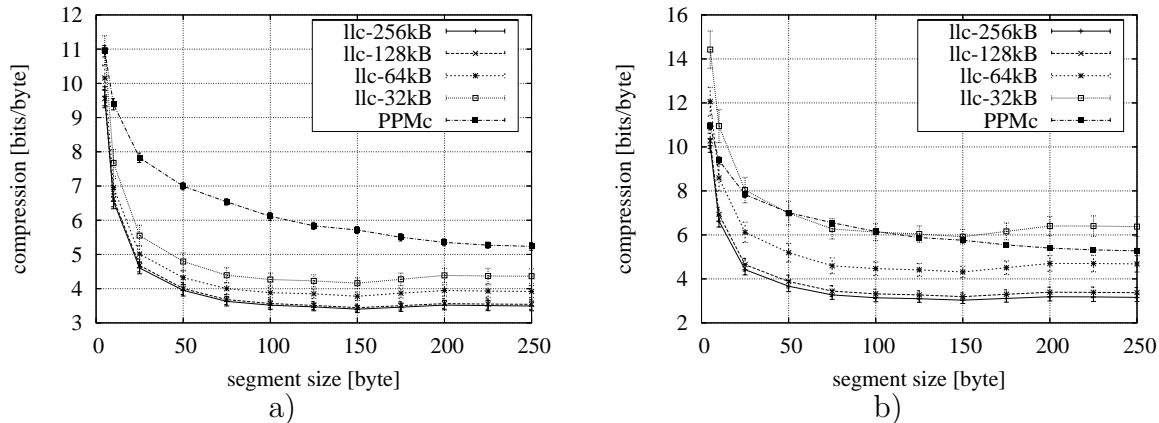


Figure 5. lcc compression performance for short messages for order 2 in Fig. a) and order 3 in Fig. b). PPMc without a preloaded model and the 32 kByte order 3 model fail for short messages. The other models indicate the ability of lcc to provide a scalable trade-off between compression efficiency and computational requirements.

performance.

5: Conclusion and Future Work

In this paper, we have detailed a methodology for lossless compression of short text messages. Most of the related work in the field of text compression has been devoted to the development of algorithms/software for compression of long data files on personal computers. Due to their complexity and their adaptive statistical model these algorithms are not applicable for short message compression on low-complexity devices.

We have detailed a specific static context model that projects highly complex statistics on a succinct data array with statistical integer numbers and parity bytes. One key component is the usage of a smart hash key that equally fills the data model. We achieve a compression performance around 3.5 bits per byte for short text sequences larger than 75 Bytes using an order 2 context model of 128 kBytes with typical training data. The compression is improved by 0.4 bits per bytes with a model order of 3. A good trade-off between compression efficiency and computational requirements is the 64 kByte model of order 2, which cuts the text sequences in half if they are larger than 75 Bytes. We note that a software provider can even significantly improve the compression performance by using training data that is typical for the users.

Our further investigations will concern the offline optimization of context models that were gathered by single training files with specific low-complexity algorithms, an analysis of other types of hash keys and their effect on the compression, the development of more effective and learning hash-keys for data modeling, and an

analysis of the compression performance for other data types.

We are planning to provide a free sample software for our short message compression method on <http://kom.aau.dk/project/mobilephone>. The software is implemented for the complete SYMBIAN OS series 60 platform. Currently we perform power consumption and complexity measurements on those architectures. Later, we will port the software towards JAVA to make it available to a larger number of mobile phones. With such a software, cell phone users may cut their costs for short message services in half.

6: Acknowledgment

We are grateful for the help of Morten V. Pedersen from Aalborg University, Department of Control Engineering, who successfully applied our method to the NOKIA 6600/ 6630 cell phones.

References

- [1] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the IEEE Data compression conference (DCC'97)*, 1997.
- [2] T. Bell, I.H. Witten, and J.G. Cleary. Modelling for text compression. In *ACM Computing Surveys (CSUR)*, volume 21, pages 557–591, Dec. 1989.
- [3] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, 1990.
- [4] J.G. Cleary, W.J. Teahan, and I.H. Witten. Unbounded length contexts for ppm. In *Proceedings of the IEEE Data Compression Conference (DCC'95)*, pages 52–61, March 1995.
- [5] M. Drinic, D. Kirovski, and M. Potkonjak. Ppm model cleaning. In *Proceedings of the IEEE Data Compression Conference (DCC'95)*, pages 52–61, March 1995.
- [6] E. Hatton. Samc-efficient semi-adaptive data compression. In *Proceedings of the IBM 1995 conference of the Centre for Advanced Studies on Collaborative research*, page 29, Toronto, Ontario, Canada, Nov. 1995.
- [7] B. Jenkin. Algorithm alley. In *Dr. Dobb's Journal*, Sept. 1997.
- [8] G. Korodi, J. Rissanen, and I. Tabus. Lossless data compression using optimal tree machines. In *Proceedings of the IEEE Data Compression Conference (DCC'05)*, pages 348–357, March 2005.
- [9] D.A. Lelewer and D.S. Hirschberg. Streamlining context models for data compression. In *Proceedings of the IEEE Data compression conference (DCC'91)*, pages 313–322, Snowbird UT, 1991.
- [10] M. Perathoner M. Hart, G. Newby. Project gutenbergliterary archive foundation. 809 North 1500 West, Salt Lake City, UT 84116.
- [11] S. Rein and C. Gühmann. Arithmetic coding—a short tutorial and free source code. Technical report, Wavelet Application Group, April 2005.
- [12] D. Shkarin. Ppm: one step to practicality. In *Proceedings of the IEEE Data compression conference (DCC'02)*, pages 202–211, April 2002.
- [13] P. Skibinski and S. Grabowski. Variable-length contexts for ppm. In *Proceedings of the IEEE Data compression conference (DCC'04)*, pages 409–418, March 2004.