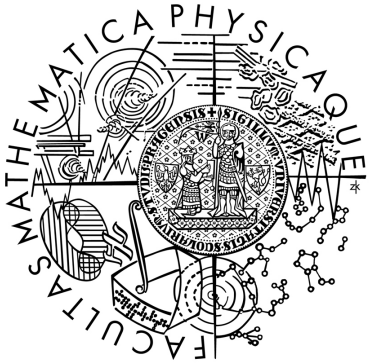
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Přemysl Paška

Context Modelling for Statistical Data Compression

Department of Software and Computer Science Education

Supervisor: RNDr. Tomáš Dvořák, CSc.
Study Program: Computer Science

2008

I would like to thank my supervisor, RNDr. Tomáš Dvořák, CSc., for his valuable advice.

Devoted to Zuzana Dotlačilová, a great person.

I declare that I have written this master thesis on my own and listed all the used sources. I agree with lending of the thesis.

Prague, August 4, 2008

Contents

1 Introduction **9**
- 1.1 Context Models . . . 9
- 1.2 Goals . . . 10
- 1.3 Structure of This Work . . . 10

2 Context Modelling Methods **11**
- 2.1 Prediction by Partial Matching . . . 11
  - 2.1.1 Evolution of PPM . . . 12
- 2.2 Context Mixing and PAQ . . . 14

3 Context Tree with the Complete History **17**
- 3.1 Simple Strategy Is Suboptimal . . . 18
  - 3.1.1 Example: Some Occurrences Are Ignored . . . 19
  - 3.1.2 Example: Long Similar Contexts . . . 20
- 3.2 Known Solutions . . . 21
- 3.3 The Ideal . . . 22
- 3.4 The Algorithm . . . 24
  - 3.4.1 Some More Situations to Handle . . . 24
  - 3.4.2 Data Structures and Terminology . . . 25
  - 3.4.3 Algorithm Overview . . . 27
  - 3.4.4 Updating a Leaf Context . . . 29
  - 3.4.5 Updating a Multi-node . . . 30
  - 3.4.6 Algorithm Summary . . . 31
  - 3.4.7 Checking the Motivation Examples . . . 32
- 3.5 Context Tree Properties . . . 34
- 3.6 Implementation . . . 38
  - 3.6.1 Obtaining Probabilities . . . 39
  - 3.6.2 Information Inheritance Approximation . . . 39
- 3.7 Empirical Results . . . 41

4 Weighted History **45**
4.1 Weighting the History 46
4.1.1 The Concept 47
4.1.2 Weight Components 48
4.1.3 Secondary Model 49
4.1.4 Counters 50
4.2 Empirical Results 51

5 Conclusion **55**
5.1 Further Work 56

A Detailed Empirical Results **57**

B Enclosed CD Contents **59**

C Compilation and Usage of the Implementation **61**
C.1 Compilation 61
C.2 Usage 62

Název práce: Kontextové modely pro statistickou kompresi dat
Autor: Přemysl Paška
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Tomáš Dvořák, CSc.
e-mail vedoucího: Tomas.Dvorak@mff.cuni.cz

Abstrakt: *Současné metody kontextového modelování používají agregovanou formu statistik a historii dat využívají jen výjimečně. Tato práce představuje dvě nezávislé metody, které používají historii sofistikovaněji.*
Metoda Prediction by Partial Matching (PPM) při aktualizaci kontextového stromu ignoruje předchozí výskyty nově přidávaného kontextu, což zhoršuje přesnosti výsledných pravděpodobností. Je představen vylepšený algoritmus, který používá kompletní historii dat. Empirické výsledky naznačují, že tato neoptimalita PPM je jednou z hlavních příčin problému nepřesných statistik v kontextech vyšších řádů.
Současné metody (zejména PAQ) se adaptují na nestacionární data pomocí silného zvýhodnění nejposlednějších statistik. Metoda představená v této práci zobecňuje tento přístup zvýhodňováním těch částí historie, které jsou nejvíce relevantní k právě zpracovávaným datům a její implementace poskytuje zlepšení na téměř všech testovaných datech zejména na vzorcích nestacionárních dat.

Klíčová slova: kontextové modely, kontextový strom, PPM, bezztrátová komprese dat

Title: Context Modelling for Statistical Data Compression
Author: Přemysl Paška
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Tomáš Dvořák, CSc.
Supervisor's e-mail address: Tomas.Dvorak@mff.cuni.cz

Abstract: *Current context modelling methods use an aggregated form of the statistics reusing the data history only rarely. This work proposes two independent methods that use the history in a more elaborate way.*
When the Prediction by Partial Matching (PPM) method updates its context tree, previous occurrences of a newly added context are ignored, which harms precision of the probabilities. An improved algorithm, which uses the complete data history, is described. The empirical results suggest that this PPM sub-obtimality is one of the major cause of the problem of inaccurate probabilities in high context orders.
Current methods (especially PAQ) adapt to non-stationary data by strong favoring of the most recent statistics. The method proposed in this work generalizes this approach by favoring those parts of the history which are the most relevant to the current data, and its imlementation provides an improvement for almost all tested data especially for some samples of non-stationary data.

Keywords: context models, context tree, PPM, lossless data compression

Chapter 1

Introduction

1.1 Context Models

A context model is the key component of every modern statistical data compression. Currently, there is no method that achieves sufficient compression performance without using a context model.

The statistical data compression method consists of two relatively independent parts: a model and an encoder. First, a statistical model is constructed based on the input data; this part is addressed by context modelling. In the second step, the input data are encoded with respect to symbol probabilities provided by the statistical model. The more probable symbols are assigned shorter codes; and conversely, the less probable symbols are assigned longer codes. That is why, the output is smaller than the input (provided that the input contains some redundancies and the model is able to detect them). The encoding step is quite straightforward and can be done using the arithmetic coding [1] or any other entropic coding.

The statistical model makes the difference between statistical compression methods. It can be as simple as a set of frequency counters, one for each symbol, but usually it consists of many context models combined together in a sophisticated manner. Context models utilize the fact that most of real data are contextually sensitive. In English text, for example, the letter "q" is most probably followed by the letter "u"; therefore, the model assigns a large probability to the symbol "u" in the context "q". More symbols than just a single one can be used as a context; it can be an arbitrarily long string or a sequence of words if we want to use word-based modes. Theoretically, any part of the input preceding the symbol currently being encoded can be used as a context.

It is important to note that the model is almost exclusively constructed on-line as the data are processed [2]. Although we could process the data

in two phases – gather the statistics in the first run and encode the data using the resulting model in the second run, this approach has several major disadvantages (the model would have to be included in the output data, etc.). In the on-line approach, only the processed part of the input data is used for construction of the statistical model. Therefore, the decoder is able to construct the model in the same way as the encoder, and there is no need to include the model in the output.

1.2 Goals

In this work, we try to explore some new directions in context modelling. When context modelling methods were new, they were considered too slow and consumed too much memory; with current computing power and memory capacity, the first context modelling methods are not a problem at all.

We try to loosen requirements on low usage of computing resources to allow examine methods that might not be usable right now, but that may be usable after applying some optimizations or approximations, or even after some future hardware will allow widespread usage of the methods.

Widely adopted modern compression methods are powerful and their implementations highly optimized, and there is not exactly a high demand for new compression methods. But data compression is by far not the only application for context modelling. Marcus Hutter says [18]: "*Being able to compress well is closely related to acting intelligently*," which is a meaningful motivation for exploring new context modelling ideas.

1.3 Structure of This Work

In chapter 2, we briefly describe PPM, the method which our work is based on, its evolution and *Context Mixing*, a newer related method.

In chapter 3, we propose an improved algorithm that maintains the data structures used in PPM, which concentrates on careful use of the complete history of the data.

In chapter 4, we introduce a method called *Weighted History*, which shifts the concept presented in chapter 3 even further.

Chapter 5 summarizes the results of both methods and suggests further directions of the research.

Chapter 2

Context Modelling Methods

2.1 Prediction by Partial Matching

The most popular context modelling method so far is *Prediction by Partial Matching* (PPM). This method inspired many others to improve it, and some of its successors were widely adopted including usage in some commercial archiving applications. The original algorithm was introduced in 1984 by Cleary and Witten [3] and quickly became the standard in statistical data compression.

The basic idea of PPM is to predict the upcoming character based on the last few characters in the input data (or bytes rather than characters for non-textual data). The algorithm uses a set of context models where each model predicts the upcoming character based on a fixed-length context string with the length varying from 0 to a predetermined maximum (the models for different contexts are often called just contexts). Such models are called finite-context models of order k where k is the number of symbols used as a context or length of the context string in this case.

Models where the context is a fixed-length sub-string from a given string (input data in our case) are also called n-gram[1] models. In case of PPM, the n-gram model corresponds to the finite-context model of order $n-1$ (an n-gram consists of a context of length $n-1$ and a character following the context).

For each character of the input data, a sequence of models is used to make the prediction. The context string of each model in the sequence is one character shorter than the context string of the previous one. The contexts of the models correspond to suffixes of the processed part of the input data with respective lengths. These models are called active models. The probabilities of the active models are combined together into a single probability distribution, and the character that actually occurs is encoded with respect to this

[1] n-gram is defined as a sub-sequence of n adjacent items from a sequence

distribution using arithmetic coding.

Each model keeps a record of all characters that have followed its context string in the part of the input data processed so far and the number of times each character has occurred. These counts are used for calculating the probability of each character. The model must be able to give a non-zero probability for any symbol of the input alphabet; therefore, it reserves some probability for the case that a new character that has not been seen so far occurs; it is called escape probability, and a virtual escape symbol is associated with it.

The probabilities provided by the active models are combined together using the escape symbols. First, the active model with the longest context is used for encoding the current character. When the character is new in the context, the escape symbol is encoded with its associated probability to signal that a shorter context must be used to encode the character. Then the next model in the sequence with one symbol shorter context is used for encoding the character, and again, it may be unable to encode it and may transmit another escape symbol. To ensure that the process ends, there is a virtual order -1 context containing all characters of the input data alphabet.

This is the basic structure of all PPM variants. Further details usually significantly differ. The maximum context order usually varies from 3 to 5, but it can also be unbounded. It has been observed that, when a high or unbounded maximum context order is used, the statistics in higher order contexts are often insufficient and the resulting probabilities are inaccurate. This problem has been addressed by many different techniques; some of them are mentioned in the next section and it is further discussed in section 3.2 on page 21.

Additionally, there are several ways to estimate the escape probabilities; they are either estimated using a heuristic technique [4] or by a secondary statistical model (this is called *Secondary Escape Estimation* and is briefly described in the next section). There are also many differences in the way of updating the statistics in individual models and many other subtle differences.

2.1.1 Evolution of PPM

When PPM was first published in 1984, it was considered powerful but consuming too much computational resources. After a series of his improvements, Moffat published a more optimized modification called PPMC in 1990 [5]. Apart from the optimizations, he improved the heuristics for estimating escape probabilities (see also [4]). The escape estimation method was further revised by Howard in his PPM variant is called PPMD [6].

Cleary and Witten explored usage of unbounded length contexts in their variant called PPM* [7] first published in 1995. They define the term deterministic context as the context which gives only one prediction (i.e., every time

the context appeared it was followed by the same symbol). They suggest to use the shortest deterministic context for prediction if there is any available. To be able to access all possible contexts and simultaneously to avoid repeated scanning back through the input, their algorithm buffers all the input, and for some contexts, it stores a link into the buffer pointing to the last occurrence of the context. The algorithm had been and improvement over PPMC, but a more recent work by Bunton [2] concluded that it did not achieve its potential.

A significant set of improvements was published by Bunton in her Ph.D. thesis [2] in 1996. Recall that, by default, the longest active context is used to make the initial prediction; with Bunton's *Information-theoretic State Selection*, a shorter context may be selected as more beneficial for the compression performance. Her concept of *Mixtures* addresses the problem of inaccurate probabilities in high order contexts by utilizing inherited probabilities from parent contexts. She also proposed modification to updating of statistics in individual contexts called *Update Exclusions*; with this adjustment, not all active contexts are updated, but only those which are used for actual coding. Each of these three enhancements notably improve the compression performance of PPM, and their combination results in a significant compression gain. Bunton also suggested to use a more economical data structure, a linear-space suffix tree. The data structures used in PPM are also briefly discussed in the introduction of chapter 3 on page 17.

Bloom contributed his own set of improvements [8], one of them similar to Bunton's *Information-theoretic State Selection* (named *Local Order Estimation*). Again, the technique is selecting a context used for initial encoding, which can be shorter than the longest active one, but it is using a different criterion. By far the biggest improvement comes from *Secondary Escape Estimation*. This technique replaces escape probability estimation using a heuristics by employing a secondary statistical model. The model uses several characteristics of the primary context (such as context order) as its own context and tracks how often the escape event happens in the circumstances defined by the secondary context. Bloom also improved handling of deterministic contexts. These improvements, published in 1998, have been incorporated into a PPM variant named PPMZ.

PPMZ was the state of the art compression algorithm, but its major drawback was slowness and high usage of memory. In 2001, Shkarin introduced [9] his own PPM variant based on PPMD aiming to reduce the complexity of modern PPM variants. The main improvement is a technique named *Information Inheritance*. It is very similar to Bunton's *Mixtures* but is less computationally expensive. The resulting algorithm named PPMII (*PPM with Information Inheritance*) also includes a careful implementation of *Secondary Escape Estimation*. PPMII gives better compression results than PPMZ, and it also

demands less computational resources (both in time and space).

PPMII has been the last significant update to the PPM method. Since then, a related but more generalized method named *Context Mixing* has been used in a series of compression algorithms called PAQ, which are currently achieving top ratings in several compression benchmarks.

2.2 Context Mixing and PAQ

In PPM, each symbol is encoded based on a prediction from a single n-gram model (which is selected using a sequence of escape symbols), and the models used are exclusively byte-oriented n-gram models. Mahoney found that limiting, and his PAQ compression algorithm [11, 12] uses many different context models, and the final prediction is computed using a weighted average of probability estimates from all of the models.

Apart from the standard n-grams models, PAQ uses many context models each specialized for a certain type of data. Word-oriented contexts improve compression of text files. Fixed-length record models are useful for modeling two-dimensional data such as images or tables. For audio data, it may be useful to discard the low order bits of the audio samples from the context, because they are usually noisy. And many other models are used in different PAQ versions.

To enable combining statistics from many different contexts, only one bit at a time is predicted; each model estimates that the next bit of the data will be 0 or 1. These predictions can be easily combined by weighted averaging (for example, there is no need to worry about escape probabilities). After a bit is encoded using the arithmetic coding, the model weights are adjusted to favor the most accurate models.

The statistics in each model are updated in a way that favors the most recent observation; therefore, a model quickly adapts to changes in non-stationary data. Each model counts zeros and ones, and when a counter is updated, the opposite one is approximately halved. For example, when the count of ones is 12, and the count of zeros is 3, and a zero occurs, the count of ones is set to $7 = 12/2 + 1$, and the count of zeros is incremented to 4.

The source code of PAQ has been released under a free license (GNU General Public License [20]), and the program is still being developed by many individuals in a collaborative effort [13]. A lot of experimentation with different context models has been done; for example, PAQ8 is able to significantly compress already compressed JPEG images. In more recent versions of the program (PAQ7 and later), the probabilities are combined using an artificial neural network.

Compression efficiency of the method is impressive; for example, a derived

algorithm is the current winner of the *Hutter Prize* [18], whose motivation lies in artificial intelligence (as mentioned in the goals of this work, section 1.2). Nonetheless, it should be noted that the method operates very slowly and demands relatively high amounts of memory.

Although PAQ is a very promising method, in our work, we propose an algorithm based on PPM. We believe that compression efficiency of a byte-oriented algorithm can be sufficient because most of real data are byte-oriented, and this approach has a great performance advantage (a bit-oriented method needs eight steps, yet simpler, to encode a byte). Furthermore, we believe our ideas can be applied also to a bit-oriented method, and there is always a possibility to use a byte-oriented model in a PAQ-like algorithm.

Chapter 3

Context Tree with the Complete History

In PPM, statistics for a large number of contexts are stored in the n-gram models. The contexts are usually organized into a tree. PPM implementations usually use a variant of the standard data structure called *trie* [7, 14]. In such tree, nodes represent individual contexts; the root node represents context of order 0, and the order of other contexts corresponds to the distance of each node from the root.

The standard edges of *trie* correspond to transitions to the next active context. Bunton [2] suggested use of suffix pointers along with the standard edges; the suffix pointers connect each context with its longest suffix and are useful for escape operations.

We use a slightly different approach; we completely abandon transition edges and use only the suffix edges; furthermore, they lead in the opposite direction. Thus, the edges can be used for finding the longest active context because the reverse of the context string of each node (i.e., the string read in the direction of growing context order) describes the path from the root to the node. We use the edges in both directions though (every search operation is recorded into an array for this purpose). Any two contexts connected with a suffix edge are also mutually referred to as parent and child (parent is the one closer to the root).

Figure 3.1 shows an example of our context tree for a modified classic example [7], string "abracadabraka"; such tree is a result of Algorithm 1 described in the next section.

This organization of the context tree is much more convenient for our algorithm. When we want to add an extension of a context to higher orders, the new context is connected to a leaf node, and when there is need to add more different extensions of the same context, they are all added to the same node

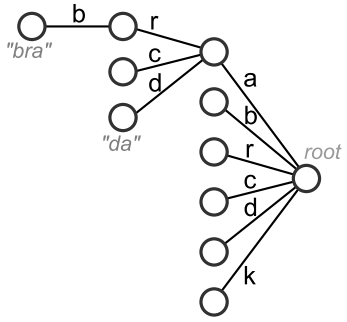


Figure 3.1: Context tree for the string "abracadabraka"

of the tree. Details are described in section 3.4 on page 24.

The information carried by the transition edges (which we abandon) must be expressed in some other way. The simplest way is to store the statistics in an array of pairs [symbol, count] in each context. Other possibilities are discussed in section 3.4.

Recall that the statistical model is constructed on-line; at the beginning of encoding (or decoding) the model is empty, and it is incrementally updated after coding of each symbol. At this point, new contexts are added into the context tree. We usually can't afford to store all contexts of all lengths because of limited resources. There are many ways how to choose which contexts to store or, in terms of the data structure, how to build the context tree. In this chapter, we describe an algorithm for building the context tree, but first, we address some sub-optimality of the standard approach. Finally note that in our descriptions of algorithms we do not include handling of escape probabilities because it is not important for our purposes.

3.1 Simple Strategy Is Suboptimal

As stated above we can't afford to add every context that appears in the data into the context tree; fortunately, there is one obvious case when employing a longer context is useless. When the context is deterministic (defined in [7], see section 2.1.1 on page 12), there is no need to further extend the context because the probability distribution would not change.

This idea can be directly applied to an algorithm for building the context tree: whenever a leaf context is deterministic, do not add any child contexts. At each update of the context tree, we start with searching for the longest matching context. Then we update the statistics in the context, and we add a new child context to it at each update except for the situation when the context is deterministic and the symbol already predicted by it occurs (adding a child

Algorithm 1 "the simple strategy"

- find a context with the longest match
- when symbol is already present in the context's statistics
  - increase counter of the symbol
- otherwise
  - add the new symbol to the statistics
- if the context is not deterministic
  - add a new child context with the new symbol to the context

context to the longest matching context is always correct, because we know that there is no child context with a context matching the current context).

Algorithm 1 shows summary of the procedure, which is called at each tree update; its input is the symbol that has occurred.

The following examples show that this strategy can lead to very inaccurate probability distributions if we do not make further modifications to the algorithm.

3.1.1 Example: Some Occurrences Are Ignored

Consider the following data sample: "hat_cat_cat_cat_cap" (the underscore symbol denotes the space character in all examples). We use it to demonstrate a case when the algorithm gives us an inaccurate probability distribution.

Figure 3.2 shows the state of the context model after processing the data sample without the last symbol (on the left). We can see that all contexts of order 1 are deterministic; that's why, it was not necessary to add contexts of higher orders. Let us now see the resulting probabilities after the algorithm updates the model when processing the symbol "p" at the end of the data sample. The longest matching context is "a". The symbol "p" is not yet present in it, so we add it. Since the context "a" is no longer deterministic, we also add a new child context "ca" and the symbol "p" into it (shown on the right of figure 3.2).

We can see that the resulting probability of the symbol "p" in the context "ca" is 1 (as stated above, we ignore any escape probability); although, when we consider all occurrences of the context "ca" in the data, the probability should be 1/4 (and 3/4 for the symbol "t"). That is because the algorithm

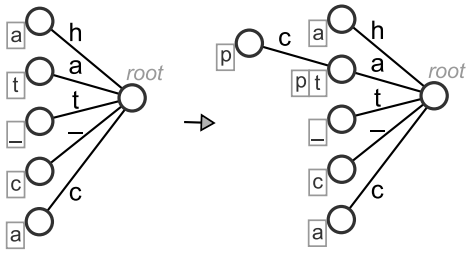


Figure 3.2: A context tree change after processing the last symbol of the string "hat_cat_cat_cat_cap". The gray rectangles contain context's alphabet.

does not take into account any previous occurrences of a newly added context.

3.1.2 Example: Long Similar Contexts

A similar problem can be observed when we consider a rare context that needs to be relatively long to be deterministic. Examples of such contexts can be found in sub-strings of phrases "an_appreciative" and "and_appreciate". A context "_appreciat", which is a common part of both phrases, predicts "i" in the first case and "e" in the second one. Consider some data where the two phrases occur alternatively, possibly with some other data in between (which do not contain the symbol "t", for simplicity of this example). Let us examine evolution of the probability distribution in the context "_appreciat" and its parent contexts ("appreciat", "ppreciat", ..., "t").

The first time the phrase "an_appreciative" is encountered, the context "t" is seen for the first time as well. According to the algorithm, we add the context "t" into the context tree (provided that the zero order context is no longer deterministic after the symbol "t" has been added into it) and the symbol "i" which follows the new context into it (figure 3.3, step 1). Then the phrase "and_appreciate" is encountered, where the context "t" is followed by the symbol "e". The longest matching context "t" does not contain "e", so we add it, and we also add a new context "at" and the symbol "e" into it (step 2). Then "an_appreciative" occurs again, where the longest matching context "at" is followed by "i". The symbol is not yet present in the context, and is added into it along with a new child context "iat", with the symbol "i" in it (step 3). The situation repeats until the phrases are encountered eleventh times (if strictly alternating). At that time, a context "n_appreciat", which deterministically predicts "i" (within our data), is added (step 11). And the next time "and_appreciate" is encountered, context "d_appreciat", deterministically predicting "e", is added into the tree (step 12).

We can see that the phrases must be encountered at least eleven times before one of the deterministic contexts "n_appreciat" or "d_appreciat" is

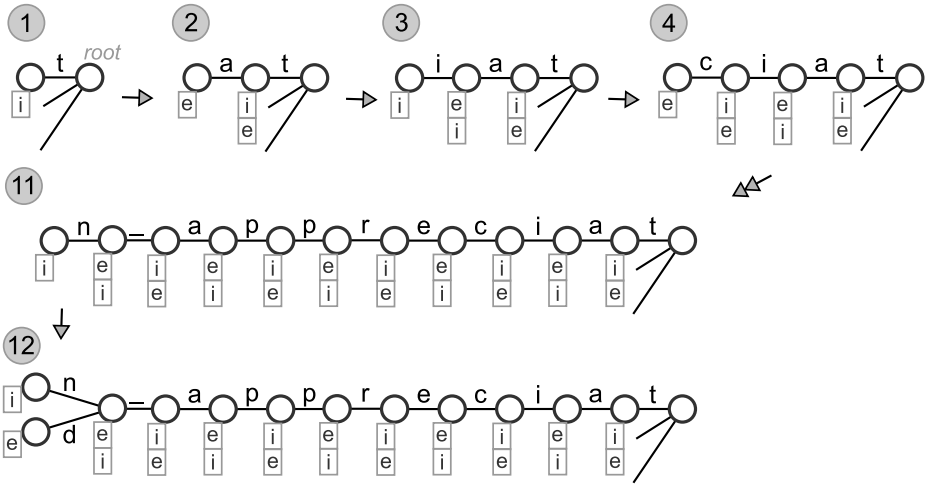


Figure 3.3: Evolution of a context tree for long similar contexts

added. And before that, only one symbol ("i" or "e") is predicted in the longest matching context when considering the context "t" and its child contexts.

If the algorithm were smarter, it would make a look-up for the previous occurrence of the context in the second step of our example, and it would be able to add a deterministic context immediately. And there is also the same problem as demonstrated by the previous example: when we consider all occurrences of any context shorter than the shortest deterministic, probabilities of both "i" and "e" should be close to 1/2 when the context is added into the tree.

In the next section, we mention some already know solutions to the problem. In later sections we describe the problem more generally and we propose our own solution.

3.2 Known Solutions

The problem of inaccurate probabilities in high order contexts gains importance with the unbounded context order. The first algorithm that does not apply fixed upper bound on context order, the PPM* by Cleary and Teahan [7], also tried to solve the problem of inaccurate probabilities in high order contexts. The model stored all unique contexts along with links to their occurrence in the input string (that required buffering of all the input). The strategy has been an improvement but it was not a great success.

With respect to our examples (sections 3.1.1 and 3.1.2), it is helpful only partially. In the first case, the PPM* algorithm would result in a probability distribution of the context "ca" with symbols "p" and "t" equiprobable. That

is because whenever a new symbol is added into a previously deterministic context, the link into the input string is used to look up the symbol that was previously seen in the context. Then both the new and the previously seen symbol are added to the context. There can be more occurrences of the context, but only the one which is pointed to by the link is considered. In the second case (3.1.2), adaptation would be much faster because the model stores all unique contexts; therefore, this situation is not a problem for PPM*.

Shkarin [9] and before him also Bunton [2] address the problem indirectly (see section 2.1.1 on page 12). In Shkarin's PPMII, when a symbol is added to a context, its count is initialized based on the probability of the symbol in the parent context.

This technique significantly reduces the sub-optimality of the simple strategy, demonstrated on the data sample "hat_cat_cat_cat_cap" (the example in section 3.1.1). Although after processing this sequence, the probability distribution in the context "ca" is the same as if using the simple strategy, next time the symbol "t" is encountered in this context, its initial count is set with respect to high probability of the symbol in the parent context "a". That has a similar effect as considering all occurrences of the context. The main difference is that the probability of the symbol in the child context may be quite different from the probability in the parent context (it is just a heuristic method). Another subtle distinction is that the probability distribution is corrected a little later when symbols ("t" in this case) from the disregarded occurrences are encountered.

On the other hand, *Information Inheritance* is not helpful in the case of the second example (section 3.1.2). The model would still need the same number of occurrences of the phrases as the simple strategy until a context long enough to be deterministic is added.

In the next section, we discuss a more general goal for an ideal algorithm for building the context tree. Then we will propose a different algorithm which fulfills the goal.

3.3 The Ideal

Ideally, we would like to gather statistics for exactly all different contexts and consider all occurrences of the contexts (i.e., not to miss any occurrences of a context before it is added into the tree). First, we must define which contexts do we consider different or, conversely, which contexts are equivalent. The second requirement of considering all occurrences will not be discussed in this section, but it will be addressed by the algorithm presented in the next section.

The output of a context model is a probability distribution; therefore, we define equivalence of contexts based on probability distribution. Note that

we consider distributions without any renormalization of counts (i.e., all occurrences are counted equally regardless of their position in the input data), which usually is applied in real implementations.

Definition 1. *Probability distribution of a context* is a probability distribution of symbols that follow individual occurrences of the context in the data where each symbol's probability is proportional to the number of occurrences it followed.

(So the facts needed to specify a probability distribution of a contexts are: which symbols occurred in the context and how many times each of them.)

Definition 2. Two contexts are *equivalent* if they have the same probability distributions on given data.

Unfortunately, this definition does not give us any clue how to recognize the equivalent contexts except perhaps by comparing the probability distributions, which is not too practical. In general, there is perhaps no easy way to do that since two contexts may be equivalent quite "accidentally". Consider the following data: "can_ban_cat_bat". Contexts "ca" and "ba" are equivalent by the above definition without marks of any other relationship.

Luckily, there are cases of equivalent contexts that are easily recognizable. For example, when we pick a context long enough to occur only once in the whole input data, we can make this context arbitrarily longer, and we always get the same probability distribution (containing the only symbol predicted by the context). This leads us to the following idea:

Proposition 1. *Consider two contexts C and S, such that S is a suffix of C. If the context S occurs only as a suffix of C, then contexts C and S are equivalent on given data.*

Consider the following data: "Betty_Botter_bought_some_butter" (a part of a tongue-twister). For example, context "te" occurs only as a suffix of "tte", and, indeed, probability distributions of the contexts "te" and "tte" are the same (they consist of two occurrences of symbol "r"). For another example take contexts "bo" and "o"; context "o" occurs as a suffix of "bo", but has another two occurrences (in "Bo" and "so"); therefore, the contexts "bo" and "o" do not fulfill the assumptions of the proposition (and, by a coincidence, their probability distributions are different).

Proof of the proposition is quite straightforward:

Proof. The fact that S is a suffix of C implies that S occurs whenever C occurs. As a consequence, when we examine probability distributions of two contexts where one occurs *only* as a suffix of another, we realize that they consist of

exactly the same symbols at exactly the same locations of the data; therefore, the probability distributions must be the same.

□

The proposition will help us save some space by storing only one copy of statistics for equivalent contexts. In the following sections, we will introduce an algorithm which was designed to reach goals presented in this section.

3.4 The Algorithm

Here we present our algorithm for building the context tree. In section 3.1, we have shown that inaccuracy of probabilities in long contexts is partially caused by ignoring occurrences of a particular context occurring before the context is added into the context tree. To correct this behavior, our algorithm stores a list of all occurrences at each leaf context; the list is used for searching for all occurrences of a child context newly added to the leaf.

3.4.1 Some More Situations to Handle

Since we want to have all different contexts in the tree, the algorithm for updating the context tree must handle several situations which are not handled by the standard algorithm. Note that storing all different contexts implies that we need all leafs to be deterministic. If a leaf is not deterministic (i.e., it predicts at least two distinct symbols) there exist at least two different extensions of the leaf context string such that each of them predicts a different symbol; thus, the context tree does not store all contexts.

As we could see in the example with the phrases “and_appreciate” and “an_appreciative” (section 3.1.2 on page 20), it is not always sufficient to extend the context by one symbol when we want to keep leafs deterministic; therefore, our algorithm extends contexts until they are deterministic.

Furthermore, to keep leaves deterministic it is generally not sufficient to add one new context for the new symbol, as it is done in the simple strategy algorithm (section 3.1 on page 18), and it is still not sufficient to add two new contexts, one for the new symbol and the second for the former one (which is what PPM* [7] does). The following example shows a situation when adding four new contexts is required to keep all leaves deterministic.

Suppose we have a deterministic context “estin” with three occurrences within the words “suggesting”, “resting” and “harvesting”; thus, the context predicts the symbol “g”. When the word “destiny” is encountered, the symbol “y” is added into the context “estin”, and it is no longer deterministic. So we add a child context "destin" and the symbol “y” into it. We should also

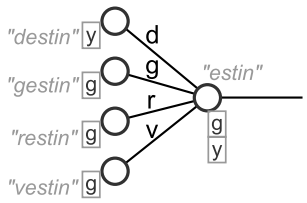


Figure 3.4: Context "estin" with four child contexts

add a deterministic context predicting "g", but should it be "gestin", "restin" or "vestin"? We must add all three to be able to distinguish them from the context "destin". Figure 3.4 shows the desired result.

This example shows that it may be necessary to add a new context for each symbol of the input alphabet after extending the context. However, in some cases, one of the new leaf contexts may still not be deterministic and the context must be further extended (this situation is discussed in section 3.4.4). In general, it may be necessary to add a new child context for each occurrence of the original context; however, that is highly improbable to happen with real-life data.

3.4.2 Data Structures and Terminology

We use a context tree as described at the beginning of this chapter (on page 17) with some adjustments. As mentioned, we add a list of links into the input string to some context nodes, and we also allow the edges of the tree to represent strings, rather than single symbols (in the following, we use the term "string" only for strings of length greater or equal to 2 when the tree edges are considered).

The list of occurrences is implemented as a list of links into the input string. We call the processed part of the input string "history" and the list "history index". For other purposes (see chapter 4), it can be useful to maintain history indexes in all contexts; for purposes of building the context tree, only history indexes in leaf nodes are required.

Nevertheless, since the history index stores all occurrences of a context, it can be used to obtain probability distribution of the context, so that any other explicitly stored symbol counts are not necessary. With respect to the algorithm described here, it is not important whether the probability distributions are obtained using a history index of from an array of symbol counts; therefore, we will use the term "updating statistics" for any of these two data structures.

Allowing edges to be labeled by strings eliminates the need to store contexts with a single child context; as a consequence, when a node is not a leaf, it has

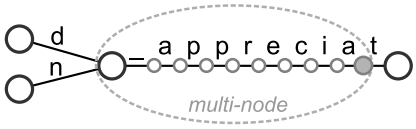


Figure 3.5: Multi-node

at least two child nodes. (Note that according to Proposition 1 a context with a single child is equivalent to the child, and this modification is thus required to avoid storing duplicate statistics for equivalent contexts.)

We call a node connected to its parent with a string-labeled edge a "multi-node" because it also represents all context nodes eliminated by using the string-labeled edge (these contexts are equivalent to the multi-node). When we want to refer to the individual nodes represented by a multi-node (including the multi-node itself) we use the term "sub-nodes"[1]. Figure 3.5 shows an example of a multi-node.

The string-labeled edge is implemented in the following way. The multi-node is connected to its parent with a standard edge (a pointer) labeled by the first symbol of the string (which is correct since the first symbol—the lowest order one—is always unique among all string-labeled edges leading from one node). The node itself contains a pointer to one of its occurrence, information about its order and length of the string; this is enough information to look up the rest of the string (these data are present in all nodes, though for most nodes the string length is equal to 1).

There are at least two reasons for this implementation. First, the algorithm recognizes equivalent contexts at a point when the lowest order context of the equivalent nodes is already present in the tree, and with this implementation changing the node into a multi-node is as easy as updating the string length. And second, if the string of an edge matches the current context only partially, we still want to use the statistics from the corresponding multi-node because the lower order contexts are equivalent to it. Therefore, it is correct that we can get into the multi-node with only the first symbol of the string matching the current context (although when this happens, the multi-node is split into two nodes during the subsequent tree update).

We refer to the situation that the string partially or completely matches the current context as "partially" respectively "fully matching multi-node". Also, when we need to point out that a node is not a multi-node, we use the term "regular node".

[1] Bunton [2] uses the term "virtual states" in a similar meaning

3.4.3 Algorithm Overview

Let us now describe the algorithm for updating the context tree step by step. Algorithm 2 shows the main part of it; its sub-procedures are shown separately. Note that adding of a child context always includes initialization of a new history index with all occurrences of it (algorithm summaries omit this for brevity) .

First, the context which has the longest match with the current context is selected; statistics of all nodes along the path from the root are updated during the search. Then the algorithm continues in one of four possible directions depending on where the selected node is situated in the context tree, and whether the current symbol is new in the context or not.

In the first case, when the selected node is a leaf already predicting the current symbol, then there is no need to add any new contexts. A new occurrence is added into the leaf's history index, and the algorithm finishes.

Everything important happens in the second case when we need to add some new child contexts to a leaf context which is no longer deterministic (because a new symbol occurred in it). This is handled by the procedure *branch_ leaf*. We will describe it in detail in the next section. The key issue is that the procedure considers all occurrences of the leaf context when the new child contexts are added.

The third and fourth cases correspond to the situation when the longest matching context is an inner node. In the third case, when the selected (longest matching) node is a regular inner node or a fully matching multi-node, then a new child context with the current symbol is added into it, and the history index of the new leaf context is initialized with a single occurrence pointing to the current context.

And finally, in the fourth case, when the selected node is a partially matching multi-node, then the multi-node is split, and a new child context is added to the lower part. This is handled by the procedure *branch_ multinode* and is described in more detail in section 3.4.5.

In both cases of updating an inner node, there is no need to care about any other previous occurrences of the newly added context (and so adding only one new child context is sufficient). We know that its current occurrence is the only one because if there was any other previous occurrence, the new child context would have been already added to the inner node being updated.

In the following sections, we describe in detail the more complex cases which were mentioned only briefly here: branching of a leaf context and splitting of a multi-node.

Algorithm 2 Context Tree with the Complete History

- find the longest matching context node
  - and update statistics of all contexts along the path from the root to the node
- if the node is a leaf predicting the current symbol
  - finish
- else if the node is a leaf predicting a different symbol then the current one
  - update the node using the procedure *branch_leaf*
- else if the node is a regular inner node or it is a fully matching multi-node
  - add a child context and the new symbol to it
- else (the node is a partially matching multi-node)
  - update the node using the procedure *branch_multinode*

Algorithm 3 procedure *branch_leaf*

- go through all occurrences of the leaf's context string (including the new one)
- extend all contexts by one symbol preceding them
  - repeat extending until there are at least 2 different symbols extending the contexts
- add a new child to the starting leaf node for each distinct extended context
- if the child leaf predicting the new symbol is not deterministic
  - branch it again using the procedure *branch_leaf*

Algorithm 4 procedure *branch_ multinode*

- find the highest order sub-node matching active context string
- split the multi-node into two nodes
  - the lower one represents sub-nodes up to the highest order matching one
  - the upper one represents sub-nodes starting from one order higher then the highest matching one
  - (if any of the new nodes includes only one sub-node it becomes a regular node)
- add a child context to the lower (multi-)node and the new symbol to it

3.4.4 Updating a Leaf Context

When we need to add some new child contexts to a leaf context, we use the history index to go through all its occurrences. The contexts at each occurrence are then extended symbol by symbol until at least two different contexts are found; when extending by more than one symbol is necessary, the former leaf is changed to a multi-node by updating the string length of the edge that connects it with its parent and increasing its order accordingly. Then we add a new child node for each distinct extension of the original context string. Each new leaf gets its own history index; this is easily done by copying and filtering the history index of their parent context.

At this point, it is not guaranteed that all new leaf contexts are deterministic. More specifically, one of the new leafs, the one which predicts the symbol newly added to the original leaf, may also predict the former symbol of the original leaf. That is because the current context predicting the new symbol could be extended by the same symbol as some of the former occurrences. We must check for this situation and repeat the whole procedure for the new leaf if the check reveals that the new leaf context is not deterministic.

Let us now demonstrate working of the procedure *branch_ leaf* (Algorithm 3) on a more complex example that comprises most situations that the procedure must handle. More precisely, it shows the need for repeated branching of a leaf context during single tree update, with more than two new leaves added at each phase. Another situation that is handled by the procedure and that is not demonstrated by this example is creation of multi-nodes (this is shown by both examples in section 3.4.7).

Suppose we have a (deterministic) leaf context “rabl” with one or more

occurrences within each of the following words: “considerable”, “tolerable”, “vulnerable”, “desirable” and “favorable”. Thus, the context deterministically predicts the symbol “e”. Further, suppose that the procedure was run after the word “miserably” appeared in the data; thus, the symbol “y” was added into the context.

After extending all occurrences of the context by one symbol, three distinct contexts are found: “erabl” that occurs within “considerable”, “tolerable”, “vulnerable”, and “miserably”; then “irabl” which occurs within “desirable” and “urabl” within “favourable”. According to the algorithm, a new leaf context is added for each distinct extension of the original context. Then the leaf context that predicts the new symbol must be checked if it is deterministic. It is the context “erabl” in our case, and the check reveals that the context is not deterministic; it predicts both the new symbol “y” and the former symbol “e”. As a consequence, the procedure must be run again for the context “erabl”. The algorithm does not require to check the remaining new leaves, and we can verify that it is a correct behavior in the case of our example; all new leaves except the one that has been checked deterministically predict one symbol—the symbol “e”.

The procedure is thus restarted for the context “erable”. All context occurrences are extended by one another symbol, and four distinct contexts are found: “derable” which occurs within “considerable”, “lerable” within “tolerable”, “nerable” within “vulnerable” and finally “serabl” within “miserably”. Thus, four new leaf contexts are added, and again, one of them must be checked whether it is deterministic. This time it is deterministic (context “serabl” predicts solely the symbol “y”), and the context tree update is finished.

We can see that 7 new contexts were added during single update, 3 in the first phase and another 4 in the second one.

3.4.5 Updating a Multi-node

Another example demonstrates working of the procedure *branch_multinode* (Algorithm 4). Consider a context tree branch along the string “listen” where orders 6, 5 and 4 are represented by a multi-node (figure 3.6). The context “listen” have at least two child contexts; they may include “_listen” (occurring e.g. within “now_listen”) and “glisten” (occurring e.g. within “glistening” or “glisten”). Suppose the procedure was run after the letter “c” within the word “existence” appeared in the input data.

The highest order sub-node whose context matches the active context string is order 5 node representing context “isten” (it is a partially matching multi-node). The multi-node is split into two nodes. The lower part includes order 5 sub-node, the longest matching context, and order 4 sub-node, remaining

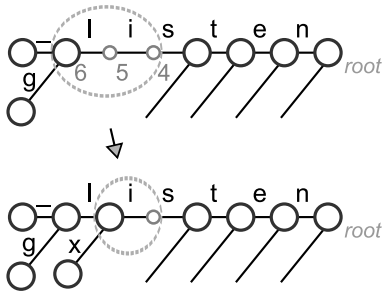


Figure 3.6: Splitting a multi-node

lower order sub-node of the former multi-node. The upper part includes order 6 sub-node representing context "listen". Since it includes only one sub-node, it becomes a regular node. A new child context "xisten" is added to the lower node of the new multi-nodes, and the symbol "c" is added into the new child context. A new history index is initialized in the new leaf context.

This was the last case of the algorithm needed to be explained. Summary of the algorithm follows.

3.4.6 Algorithm Summary

To provide a simpler view, we can say that there are just two major ways the algorithm can go if any context tree update is needed at all. A leaf context needs to be updated (the second branch of the condition in Algorithm 2), or an inner node needs an update (the last two branches of the condition). The latter case has two sub-cases which are rather technical consequence of using string-labeled edges.

The update of a leaf is handled by the procedure *branch_leaf*, which comprises most of the algorithm's complexity. The procedure recurses until all new leaves are deterministic while keeping the condition that no context occurrences are ignored.

In the next section, we verify that the algorithm works optimally in case of our motivational examples (sections 3.1.1 on page 19 and 3.1.2 on the next page). In section 3.5, we show some properties of the context tree created by the algorithm.

3.4.7 Checking the Motivation Examples

Our algorithm was designed to address some sub-optimality described in section 3.1. Let us now verify that we were successful.

The First Example

Processing of the data sample "hat_cat_cat_cat_cap" from the first example (section 3.1.1 on page 19) proceeds quite similarly to the simple strategy algorithm until the last symbol is processed. In cases of both algorithms, right before the symbol "p" is processed, the current context is "a", which deterministically predicts the symbol "t". The difference is that, in case of our algorithm, the context "a" has a history index attached to it; the index stores five occurrences of the context (including the current one). At this point, the algorithm is called to update the tree (follow Algorithm 2). We have already selected the longest matching context—it is the context "a". It is a leaf deterministically predicting a different symbol ("t") than the current one ("p"); therefore, the context node is updated using the procedure *branch_leaf* (Algorithm 3).

First, the occurrences from the history index are extended by one symbol. After the extending, two distinct contexts are found: "ha" with one occurrence and "ca" with four occurrences. A new child context is added to the original leaf for each of them. The first one, "ha", deterministically predicts the symbol "t"; the second one, "ca", has occurred four times and predicts "t" with the probability 3/4 and the new symbol, "p", with probability 1/4. And this is the important difference from the simple strategy algorithm, which would assign zero probability to the symbol "t" in the context; nevertheless, the algorithm continues because the context "ca" is not deterministic.

Note that if we extend the last two occurrences of the context "ca" (within "cap", and within the last "cat") by more than two symbols, the context strings start to overlap. Since the algorithm extends the context occurrences until they are distinct, it must reach the leading "h" to distinguish them (the resulting contexts will be "hat_cat_cat_ca" and "cat_cat_cat_ca"). The overlapping happens also in cases of the first two occurrences within "cat", though the distinguishing symbol is found sooner in these cases. The overlapping is not a problem for the algorithm at all; it has been mentioned just because the resulting context tree might look a bit too complicated at first sight.

The procedure *branch_leaf* is thus run again, this time for the context "ca". All four of its occurrences are extended by one symbol, and for all of them it is the space character ("_"). The same happens for the second and third extension, and all context occurrences are extended by symbols "t" and "a" respectively. The next extension finally distinguishes the context occurrences. The first occurrence is extended by "h", and a child node representing the

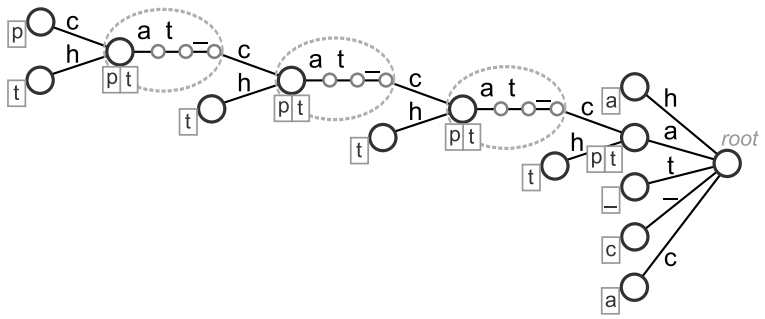


Figure 3.7: Context tree for the string "hat_cat_cat_cat_cap"

context string "hat_ca" is added to the context "ca"; since the new child context has only one occurrence, it is deterministic—it predicts the symbol "t". The remaining three occurrences are extended by "c", and a child context is created also for the context string "cat_ca". Two of the occurrences are followed by "t", and the last one is followed by "p"; therefore, the context is not deterministic, and the procedure is run again for the new child context. As more than one extension of context occurrences was needed, the node representing context "ca" is changed to a multi-node; its order is increased from 2 to 5, and it now represents four contexts ("at_ca", "t_ca", "_ca" and "ca").

For the context "cat_ca", the situation is very similar as for "ca"; it has to be extended by another four symbols to distinguish its occurrences (and the context "cat_ca" becomes a multi-node as well). Again, two child contexts need to be added: the one representing the context "hat_cat_ca" has one occurrence followed by "t", the other one representing "cat_cat_ca" has two occurrences that are followed by "t" and "p" respectively. The latter of the new child contexts must be thus branched again.

Once more, the situation is very similar. The context occurrences are extended by another four symbols (and the context "cat_cat_ca" is changed to a multi-node too). Two new child contexts are added, but this time each of them has just one occurrence. One of them represents the context string "hat_cat_cat_ca" and deterministically predicts "t", the other one represents the string "cat_cat_cat_ca" and predicts "p"; the context tree update is thus finished (figure 3.7 shows the result).

The Second Example

Recall that in the second example (section 3.1.2 on page 20), an alternating sequence of the phrases "an_appreciative" and "and_appreciate" is used as a data sample. Problem of the simple strategy algorithm is that it increases the length of contexts at most by one symbol at a single tree update; therefore, it

must process many occurrences until it recognizes the difference between such similar contexts.

Our algorithm extends contexts until they are distinct (during a single tree update); that is why, the similar contexts from the example are distinguished immediately. Let us describe it in detail. We are focusing on the context "t" and its child contexts (transitively). When the phrase "an_appreciative" is encountered, the context "t" predicting "i" is added to the tree. Then the phrase "and_appreciate" occurs in the data; thus, the context "t" has now two occurrences, one in each of the phrases. The algorithm extends the context occurrences symbol by symbol until a difference is found. The first extension prep-ends the symbol "a" to both of the occurrences, then they are extended by "i", then by "c", and so on. Finally, the tenth extension distinguishes the contexts; the first occurrence is extended by the symbol "n", and the second is extended by "d". Two deterministic child contexts are added to the context "t": the context "n_appreciat" predicts "i", and "d_appreciat" predicts "e". The context string "t" needed to be extended by ten symbols; therefore, the corresponding context node becomes a multi-node. Figure 3.5 on page 26 shows the resulting context tree.

3.5 Context Tree Properties

In the previous section, we have demonstrated that the algorithm fulfills at least some of our expectations, but can we say something more general about the algorithm? In this section, we show some general properties of the resulting context tree created using incremental updates by the algorithm.

In section 3.4.1, we emphasized that we want to have all leaf contexts deterministic, and we succeeded.

Proposition 2. *All leaf nodes in the resulting context tree represent deterministic contexts.*

For the purpose of the following considerations, we suppose that at the beginning of the coding process we have a context tree consisting of just the root node (which represents the zero-order context) with no occurrences, and we consider it trivially deterministic (it does not predict any symbol, but it will predict the first symbol of the data after it is processed).

Proof. At the beginning the proposition holds. A new leaf node can be added either during update of an inner node or during update of another leaf node. In case of the inner node update, the new leaf node always represents a deterministic context because it has only one occurrence.

The second case—a leaf update—is also the only place where a leaf can lose the property of being deterministic with addition of a new symbol. During the leaf update, all new leaves are either added being deterministic, or they are immediately updated again (by adding some new child contexts), which is also true for the original leaf.

Therefore, when the context tree update is finished, all leave nodes are still deterministic.

□

Further, we wanted that no occurrences of a context are ignored, which is one of our original goals (see section 3.3 on page 22). In the previous section, we have already demonstrated the property in one special case; nevertheless, it works generally.

Proposition 3. *Each context node contains a record about all occurrences of the corresponding context string that are present in the data.*

In other words, we want to know that any probability distribution provided by a context was estimated based on all occurrences of the context that are present in the part of data processed so far (see Definition 1 on page 23), and it does not make a difference whether we store the statistics in each context and update them at each its occurrence, or we store and update a history index that we use to look up the actual statistics later; we just want to be sure that we do not miss any context occurrence.

Proof. When a context is already present in the tree, its updating is very easy; as easy as going through all contexts along the path from the root to the longest matching context. Therefore, it remains to show that a context does not miss any occurrences before it is added into the tree. First, we will consider leaf nodes.

The root node is initially a leaf, and its history index stores every possible occurrence[2]. Whenever an order 1 leaf is added to the tree, the parent's history index is used to look up all occurrences of the new leaf, and it works the same way when a child context is later added to the new leaf (note that all occurrences of a new context can surely be found among occurrences of its parent). Since this works at any level of the tree, history index of each leaf node contains all occurrences of the context.

And it guarantees the property also for all inner nodes because we can get all occurrences of any inner context node as a union of occurrences of all its child contexts, and since every parent node was added to the tree sooner (or at

[2]In a practical implementation, it is sufficient to use history indexes for order 1 and higher, but it is an implementation detail which does not change the idea (we can consider the history buffer itsef a history index for order 0).

the same time) than any of its child nodes, all occurrences of the child contexts were considered during updates of the statistics in the inner node.

□

We also want to be sure that representation of several contexts by a single context tree node is correct.

Proposition 4. *All contexts represented by a multi-node are equivalent.*

Proof. A multi-node is created if and only if a leaf which is no longer deterministic is updated and all its occurrences are extended by the same symbol; then the leaf is changed to a multi-node. Extending all occurrences by the same symbol means that every time the context string occurred in the data, it was a suffix of the same context. According to Proposition 1, the original context and its extension are equivalent, and it is the same when the occurrences are extended multiple-times by the same[3] symbol.

When some of (higher order) contexts represented by a multi-node do not match the current context during an update, the multi-node is split, so that only contexts which really occurred at this point are represented by a single node, and those which didn't occur are represented by another one; otherwise, all contexts represented by a multi-node occurred, and they are still equivalent.

□

Apart from Proposition 3, our goal (section 3.3 on page 22) was that every possible context is stored in the context tree (Proposition 5) under the condition that only one representative for a set of equivalent contexts is stored (Proposition 6). But before showing these properties, let us formalize one quite obvious fact. Note that by extension of a context we denote a context whose context string contains the context string of the original one as a suffix (it would be a child node (transitively) of the original node in the tree).

Lemma 1. *If a context is deterministic, the context and all its extensions are equivalent.*

Proof. All occurrences of any extension of a context are also occurrences of the original context, and since all occurrences of a deterministic context are followed by the same symbol, the probability distribution can't change. Therefore, the contexts are equivalent according to Definition 2.

□

Proposition 5. *The context tree stores all contexts, but only some members of each set of equivalent contexts.*

[3] though usually different in individual extensions

In other words, it stores all unique contexts. Proposition 6 clarifies that there is exactly one member of each set of equivalent contexts.

Proof. For any leaf context, the context tree stores all shorter contexts (its suffixes). Although some of them can be represented by a multi-node; according to Proposition 4, contexts represented by a multi-node are equivalent.

Further, when a leaf is changed to an inner node, child nodes for all its extensions are added to it (see Algorithm 3 on page 28). And also when an inner node is updated, a child node is created for any new extension of the context (see listings Algorithm 2 and Algorithm 4). In both cases all occurrences of the inner node are taken into account (Proposition 3).

Therefore, if there was a different context not stored in the context tree, it would have to be an extension of a leaf context. But since all leaf contexts are deterministic (Proposition 2), again respecting all occurrences, any extension would be equivalent (Lemma 1).

□

Proposition 6. *If there are two distinct nodes representing equivalent contexts in the context tree, the contexts do not fulfill assumptions of Proposition 1.*

In other words, the context tree stores only one representative for each set of equivalent contexts, and if there are some nodes with the same probability distribution, it is just a coincidence (we are not able to recognize them with the help of Proposition 1).

Proof. Note that if two contexts are different, they will never become equivalent in future, because equivalent contexts must share all occurrences. As a consequence, when we are adding a new child context to an inner node, it can't be equivalent with its parent context; equivalent contexts can be thus added only during update of a leaf.

If all occurrences of a leaf are extended by the same symbol, the extension and the original context are equivalent; in such case, they are represented by a multi-node. The multi-node remains unchanged until some of the lower order contexts represented by the node occur separately (i.e., the multi-node matches only partially).

As a consequence, if there are two contexts with the same probability distributions represented by different nodes, they must have different occurrences, and so they do not fulfill assumptions of Proposition 1.

□

3.6 Implementation

In our implementation of the algorithm, performance and memory efficiency were not our main concerns; instead, we tried to implement many parts of the program in a universal and modular way to allow easy experimentation. Some basic optimizations were done only if tests became extremely slow.

The most relevant part of the program with respect to this work is the context tree. Its basic structure is described at the beginning of this chapter and in section 3.4.2 on page 25. To summarize, we use a context tree with edges connecting each context with its longest suffix, called parent context; the edges lead from the parent to its child. We allow the edges to be labeled by strings (longer than one symbol), and the nodes at the end of these edges are called multi-nodes, which are described in section 3.4.2 including an implementation overview. Let us now describe the data structure more in detail.

Each node of the context tree stores mainly the following items:

order the order of the context

multiNodeExtra indicates if the node is a multi-node: zero value means regular node, greater values correspond to the *string length - 1* of the edge connecting the multi-node

contextOccurrence *or* historyIndex the node has access to all of its occurrences or at least to one of them depending on which of this items is active (it is indicated by a flag also present in the node)

childKeys *and* children two arrays representing the tree edges; the first one lists symbols labeling the edges (the lowest order symbol for string-labeled edges), the second one lists the respective pointers leading to child nodes

alphabet an array of symbols that has appeared in the context (useful for quick decision whether a symbol is predicted by the context or not)

escapeCount a counter for the escape event (for PPMD-like escape method)

and a few other unimportant items. The history index is implemented as a linked list with growing number of items within its individual nodes. The arrays are implemented as dynamically allocated arrays reallocated to double size if they get too small.

The context order is limited to 255; since only very few contexts reach this limit on our test data, it practically means unbounded length contexts on most of real-life data. The order can be limited to any value between 3 and 255 using a compile-time parameter.

3.6.1 Obtaining Probabilities

The context node does not store symbol counts; the probability distribution is calculated by going through all occurrences of the context. That is very slow, but it allows easier experimentation with the way the probabilities are estimated, which we utilize in the next chapter (starting on page 45).

The occurrences are processed in the order they appeared in the data. When a symbol count reaches certain limit, all counts are scaled down by subtracting 1/4 of their value (also called renormalization), which provides better results than the usual halving. The limit can be set as a compile-time parameter, and its optimal value has been empirically estimated to be 460 (an average result on the files of the Calgary Corpus and some small samples of non-stationary data were used for these and all other similar estimates).

There are four more compile-time parameters for fine-tuning of the escape method, which correspond to a slightly generalized method *D* from PPMD [6]; these are (with their empirically estimated optimal values): initial symbol count (1), symbol count increment (1), initial escape count (7/8), escape count increment (9/8).

To speed up the processing or the history index, it is possible to process only limited number of the most recent occurrences. Since longer contexts (which influence the resulting probability distribution the most) usually have small number of occurrences, they may not be affected at all when the limit is set to a sufficiently high value. Results presented in this chapter were obtained with this limit set to $2^{16}(= 65536)$, which virtually does not affect compression performance.

3.6.2 Information Inheritance Approximation

We have implemented an approximation of Shkarin's *Information Inheritance* to allow better comparison with PPMII (see the results in the next section). The approximation blends counts from several active contexts with weights decreasing from the longest context towards the shortest one. These active contexts are not necessarily of adjacent orders because of multi-nodes; therefore, we call individual orders levels. Up to seven levels (it is a compile-time parameter) are blended this way if there is enough levels above order 1 (inclusively); however, the improvement over using four levels is negligible (below 0.003 bpB or 0.05% on average on the files of the Calgary Corpus [15]).

The calculation starts at the lowest level by gathering symbol counts from the history index of the corresponding context (including the renormalization). After the counts from the next higher level are obtained, the counts for corresponding symbols from the lower level are added to the new counts with a small weight; counts for symbols that have not appeared in the higher level

context are zeroed (i.e., the alphabet is restricted to the symbols predicted by the higher level). The resulting counts are then used in the next level, and this process continues until the longest active context is reached. As a result, only symbols predicted by the longest active context end up in the final probability distribution, and also the escape count from this context is used.

Each symbol count is thus a weighted sum of counts from several levels:

$$c = c_0 + w_1 c_1 + w_2 c_2 + w_3 c_3 + ... + w_n c_n,$$

where c_0 denotes the count from the highest level context, c_1 the count from the second highest level, etc (the addends are calculated starting from the last one, $w_n c_n$, in the actual implementation). To reduce the number of parameters, the weights are defined in a way that allows to change the formula to:

$$c = c_0 + w_1 c_1 + w_1 w_{2'} c_2 + w_1 w_{2'}^2 c_3 + ... + w_1 w_{2'}^{n-1} c_n.$$

The optimal values for the weights were estimated to be $w_1 = 1/15$ and $w_{2'} = 1/4$.

Adjustments of the other parameters are required for optimal compression performance when blending is enabled. A convenient value for the scale down limit is 300 for the longest active context and 72 for all lower level contexts. Optimal values for initial symbol count and its increment and for initial escape count and its increment were estimated to be 5/8, 1, 7/8 and 5/4 respectively.

3.7 Empirical Results

In section 3.5, we have shown that, in theory, we had achieved our goals; it remains to be seen whether the empirical results correspond to the theoretical results. Our implementation of the algorithm has no big ambitions to compete in terms of absolute compression performance; we mainly need to verify that the algorithm successfully addresses the problem of inaccurate probabilities in high order contexts. The usual test of the problem is to observe evolution of the compression performance depending on growing maximum context order, and that is exactly what we will do.

In the following, we refer to the implementation of our algorithm as *Context Tree with the Complete History*, or CTCH for short. We will compare it to PPMD [6] and PPMD with *Information Inheritance* (including *Update Exclusions Modification*) added as reported in [9] (i.e., PPMII without SEE, see sections 2.1.1 on page 12 and 3.2 on page 21); the results for both programs were taken[4] from [9]. In terms of features, CTCH is on a par with PPMD; it uses analogical escape method, *Full Exclusions* and a simple renormalization of counters. The main difference is, apart from the algorithm for building the context tree, that the probability distributions are calculated by going through a history index at each step of the coding process.

Our algorithm tries to solve the same problem as PPMII—inaccurate probabilities in high orders. So what happens when we use *Information Inheritance* on the top of CTCH? Since CTCH calculates probabilities differently, we were only able to approximate *Information Inheritance* by use of blending. For each symbol of a particular context, counts of the symbol from several active contexts are summed with weights decreasing from the longest context towards the shortest one. That is, the count from the parent is added with a small weight to the count of each symbol already present in the context; no new symbols are added (just like with *Information Inheritance*), and before that, the parent's count is augmented by the count from its parent (with even smaller weight), and so on. Details are described in section 3.6.2.

Thus, the main difference from *Information Inheritance* is that parent's statistics are used every time a probability distribution is calculated, not just when adding a new symbol to a context; therefore, it is not an exact simulation of the technique. Our approach is also much slower, but it is intended for experimental purposes only. We refer to this modification as CTCH+II.

Figure 3.8 (see appendix A on page 57 for exact data) shows compression performance (in bits per Byte; bpB) of the four methods for different maximum lengths of contexts; it is an unweighted average on all 14 files of the Calgary

[4] Although the source codes of the PPMII implementation are available, the way how to deactivate SEE is not obvious.

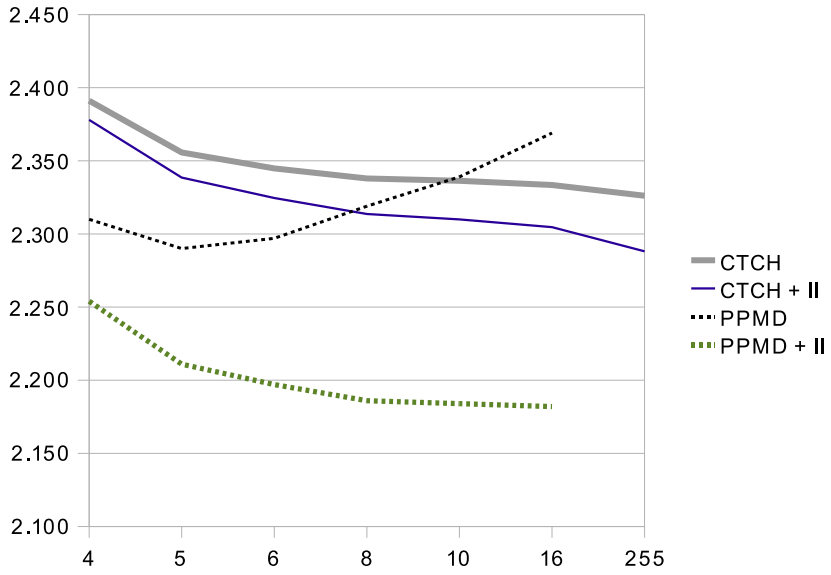


Figure 3.8: Compression [bpB] vs. maximum order

Corpus [15]. While PPMD's performance degrades when the maximum order exceeds 5, CTCH continues to improve very similarly to PPMD+II with an improvement even when the maximum is shifted from 16 to 255, which means virtually unbounded length contexts ([9] does not provide PPMD(+II) results for order 255, and the available PPMII implementation [10] does not support it either). For CTCH+II, this trend is even stronger, but the absolute improvement over CTCH is not very significant. This may suggest that CTCH itself does most of the "work" done by *Information Inheritance*.

Figure 3.8 shows that the compression performance improves with growing maximum context order, which was one of our goals. Since the Calgary Corpus comprises various types of data, it is a natural question whether the high context orders are convenient for all of them. For this purpose, we have selected some groups of files from the corpus with specific type of data:

Plain text includes *book1, book2, paper1* and *paper2*

Binary objects includes *obj1* and *obj2*

Source codes includes *progc*, *progl* and *progp*

Semi-structured text includes *bib*, *news* and *trans*

The results[5] are shown at Figure 3.9 (see also appendix A). We can see that for text files a fixed maximum order is more useful even with CTCH; the

[5]Results for "Binary objects" were multipied by 0.65 to fit the range.

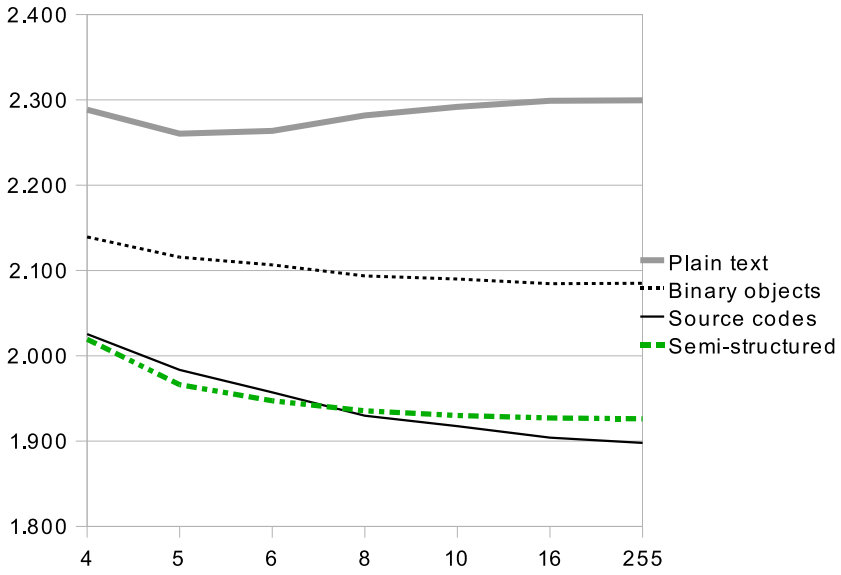


Figure 3.9: Compression [bpB] vs. order, different types of data

optimal order is around 5 or 6, but we estimate that the compression degrades much slower than it would with PPMD (compare with Figure 3.8). For other types of data, compression significantly improves with the growing maximum order; the data from the group "Source codes" exploit the long context the best.

The trend of improving efficiency with the growing order with use of CTCH is comparable to the one of PPMD+II, but the absolute compression performance is not; regrettably, we have no exact explanation for it. We suspect that there are some implementation differences that cause the degradation of compression, which is most evident at low context orders.

Unfortunately, we are not able to precisely measure the speed of the context tree update algorithm because the way CTCH obtains the probability distributions (by iterating over the history indexes) is itself very time consuming (and could be replaced by much faster convential symbol counts). Compression or decompression of *book1* by CTCH takes about 20 seconds, and the program requires up to 100 MB of memory during the coding (compared to 0.3 seconds and 16 MB required by PPMII [10] with options -o16 -m256), but we believe that most of the time is consumed during the obtaining of probabilities.

Chapter 4

Weighted History

Early PPM variants had problems with high memory requirements because at that time, typical size of computer memory was smaller by several orders than today; therefore, implementations carefully tried to save as much memory as possible. Symbol counts were usually stored in one byte each, which limits maximum value of the counts to 255. In low order contexts (especially in the zero order context), the symbol counts reach this maximum very quickly; that can be easily solved by occasional renormalization of the counts (i.e., all the counts are halved, or multiplied by a constant lesser than one). Nevertheless, when the counts are updated after the renormalization, they are still being incremented by one at each step, so that the new updates have greater impact than the old updates that were scaled down. Although it seems that renormalization of counts introduced some inaccuracy into the statistics, it has been observed that it actually improved compression performance.

The reason for the improvement is that more recent statistics are usually more relevant. In a text, for example, there may be different topics discussed, and the vocabulary used in the last few paragraphs will be used with a greater probability than the one from some more distant paragraphs. This phenomenon has been studied by many authors one of them being Mahoney [11, 12]. He criticized PPM (all variants in general) that it adapts too slowly on changes in the non-stationary data, and his method PAQ (described in section 2.2) uses special non-stationary counters to overcome this drawback.

Both renormalization of counts and non-stationary counters are variants of favoring recent statistics. This is a heuristic method that works in most cases, but there are certain situations when favoring some other parts of the data than the most recent one should be better for the overall compression performance. An example of such situation can be a stream of data where a text is interleaved by chunks of some binary data, or a text where a topic is visited repeatedly with some less related text in between.

The questions are whether we are able to recognize such situations, and if

the improvement would be worth it. In this chapter, we describe some simple and some more elaborate heuristic techniques which address these questions. In the following sections, we suggest our solution, and in section 4.2 we provide its results, which show some potential of the method.

4.1 Weighting the History

In the previous chapter, we have introduced a context tree variant that stores all occurrences of each context in a "history index" (see section 3.4.2 on page 25). The indexes in leaf nodes are used by the algorithm for building the tree, but they can be also used for obtaining a probability distribution of any context (for this purpose, they need to be maintained in all contexts). More than that, this way of obtaining probabilities enables us to do exactly what we would like to: generalize favoring of the most recent occurrences by considering relevance of each occurrence individually. In other words, we are able to assign a weight to each individual occurrence of a context, so that each symbol count is a weighted sum rather than just a simple sum with renormalizations.

The problem is how to estimate the weights. First, we need to find some indicators of the context occurrence relevance. Let us list some properties of an occurrence that might be helpful for estimating its weight:

age (i.e., the distance of the point in the data where it has occurred from the current position) the only indicator used in the standard approach of favoring the most recent occurrences

file where it occurred an easy to recognize property for multi-file archives

relevance of the history in the neighborhood this is the key property; it is much harder to estimate, and overall more complicated

Age of occurrences is a property that has been proved to work, and it is also a property that can by used for any type of data in all situations; therefore, it should be included in any algorithm for history weighting for the case that everything else does not give good results. In contrast, file where an occurrence appeared is quite application specific property, and we use only a simple weight estimation based on it.

We would like to use the last property from the list to detect parts of the history where a similar topic "is discussed" as in the part of the data being processed (the primary motivation is for text files, but it may work for non-textual data just as well). A topic can be characterized by some keywords (or some specific n-grams); we can try to estimate relevance of an occurrence by searching for some keywords that appear in the current part of the data in the neighborhood of the occurrence.

4.1.1 The Concept

Let us now describe a concept of an algorithm based on the above-mentioned ideas. Then we suggest some simplifications to make its implementation feasible.

It is impractical and computationally expensive to consider relevance of each occurrence individually; therefore, the algorithm splits the history into sections which are then "ranked" instead of the individual occurrences. The section rank is then used as one of parameters when the weights of occurrences present in the section are estimated. Ideally, the sections should be defined with respect to the natural character of the data; for example, paragraphs in text, or procedures in source codes, and subsequently adjusted to respect a predefined maximum and minimum sizes. Note that the splitting into sections must be made incrementally as the data are processed on-line.

In the second phase, these sections are ranked according to their relevance to the current part of the data, which can be defined as a concatenation of a few last sections. First, some keywords (or n-grams) of the current sections are selected, then the sections are ranked according to frequency of occurrence of the keywords. This ranking phase is performed repeatedly at least once for each new section; the old rank can reused by combining it with the new one (in a way favoring the new one).

The presented algorithm concept is quite inexplicit and raise some implementation questions. One of them is how to make the section splitting. The proposed way is hard to implement generally (paragraph boundaries differ among text formats, etc.); therefore, we will only use fixed sized sections.

Even more difficult is the selection of keywords; as a heuristic method, we will use long contexts instead. Consider a situation when a context predicts two different symbols; some of its occurrences predict the first one, and the others predict the second one. After one of the symbols is coded in the context, the occurrences predicting it can be attributed as "successful" and the others as "unsuccessful", and similarly can be classified the corresponding sections where the occurrences are located, so the successful sections are ranked higher than the others. Only long contexts can be used for this purpose because short contexts usually have a lot of occurrences in almost every section, and it would not differentiate the sections.

The following sections describe the algorithm based on the streamlined concept in a greater detail.

4.1.2 Weight Components

Each occurrence is assigned a weight estimated based on the three parameters listed in section 4.1. One weight component is calculated for each of these parameters, and the components are then combined into the resulting weight.

The age component is calculated using an inverse temporal model (discussed in [11]):

$$w_A = C \cdot D/(a + D),$$

where a denotes age of the occurrence (the distance in bytes); C and D are convenient constants (C allows to control maximum weight, and D improves behavior of the function when a is close to zero). This model performs better than others we have experimented with (linear and inverse exponential models), and it is also trivial to implement.

The history section component (listed in section 4.1 as "relevance of the history in the neighborhood") is calculated in a more complicated way; it has two sub-components, which are added up to form the section weight component. The first one is calculated using an adaptive counter (described in section 4.1.4 on page 50) stored in each section, and the second one is obtained from a secondary model (SHWE), which is quite complex (and is described separately in the next section):

$$w_S = S \cdot w_{section} + H \cdot w_{SHWE},$$

where S and H are constants that allow to change amount of influence of each sub-component at compile-time. Both the section counter and the secondary model are updated after processing each symbol using the concept of successful and unsuccessful occurrences for the longest and the second longest matching contexts. We call the section counter also primary section counter when a more precise distinction is necessary. As mentioned, all sections have the same fixed size (a convenient size was estimated to be around 350 bytes).

The file weight component (w_F) is fairly simple; its value is 1 for occurrences within the current file and a lesser constant for occurrences appearing in some of the previous files (a convenient value used in our tests is 1/4).

These three components are combined into the occurrence weight in a mostly multiplicative manner:

$$w = B_0 + w_F(E \cdot w_S(w_A + B) + w_S + w_A),$$

for convenient constants B_0, B and E. This formula is a result of experimentation with multiplicative and additive combination of the components.

We have observed that for optimal compression performance, the parameters C, S, H and B should be configured separately for different groups of orders of the primary context (the context for which we need the probability

distribution). For example, for orders 4 and higher, the secondary model (parameter H) should have much greater influence than for lower orders, especially order 1 (optimal values for high orders were estimated to $C = 184$, $S = 232$, $H = 216$ and $B = 56$; optimal values for order 1 were estimated to $C = 256$, $S = 200$, $H = 28$ and $B = 24$). On the other hand, the remaining parameters can be the same for all orders (convenient values are $D = 2^{11} = 2048$, $B_0 = 8$ and $E = 1/2^7 = 1/128$).

4.1.3 Secondary Model

Secondary models have been successfully used for estimation of escape probabilities (*Secondary Escape Estimation*, [8, 9], sections 2.1.1 and 3.2) and for correction of symbol probabilities (*Secondary Symbol Estimation*, [12]). We do not need to estimate probabilities, but there is still a reason to use a secondary model. The section counter has a significant disadvantage; its statistics are usually used only for a short time period when contexts having occurrences located in the section occur in the part of the data being processed, and then the counter becomes useless. With a secondary model, we can classify the sections into some classes, and use the secondary statistics of the classes for the whole time of the coding process. We call this model *Secondary History Weight Estimation* (SHWE) model.

To use a secondary model, we need to construct a context based on some properties of a section that determine the relevance of the section. The most important property of a section is how many occurrences of recently seen contexts are located in the section (and how many of them were successful). For this purpose, each section holds three[1] counters called *access counters*, one for each of the most recent sections. The access counter corresponding to the current section is updated in the same way as the primary section counter (according to successful and unsuccessful occurrences), and when a new section is started to be processed, all access counters are shifted, so that the oldest one is removed, and a new one for the new section is added.

Each of these three access counters is quantized into 5 bits, and the context is supplemented by one bit indicating whether the section belongs to the current file or not. This gives $2^{16} = 65536$ SHWE contexts in total; each of them stores a counter which provides the second sub-component of the section weight component (w_{SHWE}). Also this counter is updated analogically to the primary section counter and to the access counters.

[1] The number is configurable at compile-time.

4.1.4 Counters

In our implementation of *Weighted History*, we use two types of counters that differ in the way they are updated. Both types hold two values c_0 and c_1 (in our case, representing hits and misses, or in other words, successful and unsuccessful occurrences) that are used for calculating the weight:

$$w = c_1/(c_0 + c_1)$$

The first type of counter is updated in a standard way, and after that both values are scaled down to make the counter more adaptive. For example, c_0 is updated as follows (and c_1 analogically):

$$c_0 \leftarrow c_0 + i_0,$$

where i_0 is an increment. Both values are subsequently scaled down:

$$c_j \leftarrow c_j - c_j/2^K,$$

for $j = 0, 1$; K is a convenient constant set at compile-time for any use of the counter separately. The increment can be different for each of values c_0 and c_1 because there are many more unsuccessful occurrences than the successful ones, and different increments help avoiding extreme values of the weight.

This type of counter is used for the access counters (with $K = 7$, $i_0 = 8$, $i_1 = 6$) and for the SHWE counters (with $K = 10$, $i_0 = 496$, $i_1 = 160$).

The second type of counter is updated in a more adaptive way similar to the one used in PAQ (see section 2.2 on page 14, [12]). It adds another update step after the increment step. If the opposite value than the one being incremented is relatively high ($c_1 > 1/4 \cdot c_0$ when incrementing c_0), it is scaled down once more before scaling both values (the final scaling is important also for preventing an overflow):

$$c_1 \leftarrow c_1 - c_1/2^{K-4},$$

for $K \geq 5$ (and symmetrically when incrementing c_1).

This type of counter is used for the primary section counter (with $K = 11$, $i_0 = 384$, $i_1 = 44$).

4.2 Empirical Results

During the development of *Weighted History* (WH), we have found out that simple renormalization of counts works surprisingly well in comparison with our complex algorithms; therefore, our first concern was that WH should improve compression performance for most types of data. As a simple check of the concern, we use all 14 files of the Calgary Corpus [15].

To enable a fair comparison with PPMII [9, 10], we have reimplemented a simplified version of its *Secondary Escape Estimation* (SEE). PPMII uses three types of SEE for different types of the primary contexts. We have reimplemented the type used for deterministic contexts (denoted binary contexts by Shkarin), generalized it by adding the size of the context's alphabet into the SEE-context and used it for all contexts. The resulting algorithm, which also utilizes the II approximation described in section 3.6.2 on page 39, (referred to as CTCH+II+SEE) is used as a baseline for evaluating the effect of *Weighted History* (the version with WH added is denoted as +WH in the tables).

File/Algorithm	PPMII	CTCH+II+SEE	+WH	Difference
bib	1.725	1.822	1.824	0.1%
book1	2.188	2.227	2.209	-0.8%
book2	1.830	1.870	1.841	-1.6%
geo	4.335	4.565	4.537	-0.6%
news	2.188	2.281	2.216	-2.9%
obj1	3.532	3.828	3.805	-0.6%
obj2	2.161	2.280	2.221	-2.6%
paper1	2.190	2.279	2.257	-1.0%
paper2	2.175	2.235	2.225	-0.4%
pic	0.756	0.736	0.722	-1.9%
progc	2.201	2.316	2.287	-1.2%
progl	1.438	1.535	1.506	-1.9%
progp	1.451	1.523	1.506	-1.1%
trans	1.218	1.305	1.293	-0.9%
Total	1.887	1.946	1.917	-1.5%
Average	2.099	2.200	2.175	-1.2%

Table 4.1: Effect of *Weighted History* on the files of Calgary Corpus

Table 4.1 presents the results (in bpB); for reference, it contains also results for PPMII (run with arguments for maximum compression: -o16 -m256); the last column shows the effect of *Weighted History* as the relative difference between the CTCH baseline and its WH enabled variant.

All files except *bib* are compressed better by the version with WH enabled.

File/Algorithm	PPMII	CTCH+II+SEE	+WH	Difference
texts	1.998	2.018	1.959	-2.9%
CC solid shuffled	1.934	1.967	1.908	-3.0%
CC solid	1.929	1.970	1.915	-2.8%
txt-src	1.546	1.588	1.535	-3.4%
kennedy.xls	1.344	1.187	1.145	-3.5%
world192.txt	1.211	1.276	1.274	-0.1%

Table 4.2: Effect of *Weighted History* on archives and some large files

The improvement is significant especially for files *news, obj2, pic, progl* and *book2*; thus, for largish files in general (with one exception being *progl*). Unfortunately, PPMII is better than both CTCH variants on most files for an unknown reason (which is probably hidden in the fact that CTCH is worse than PPMD at low orders).

Our motivational use cases involved some non-stationary data where a specific type of data appears repeatedly. We have tested such situations with help of support for multi-file archives in our implementation (it processes the files in the order given by the command-line arguments). Since the PPMII implementation does not support multi-file archives, we provide results for concatenations of the corresponding files in the respective order.

Table 4.2 shows the results for the following items:

texts a collection of all non-binary files from the Calgary Corpus in the following order: *news, bib, book1, book2, paper1, paper2, progc, progl, progp* and *trans*

CC solid shuffled a collection of all files from the Calgary Corpus shuffled, so that different types of data alternate; the order is: *paper2, news, bib, progc, geo, progp, obj2, book1, pic, progl, obj1, book2, trans* and *paper1*

CC solid the same collection of files in an order that groups different types of data together: *news, bib, book1, book2, paper1, paper2, progc, progl, progp, trans, obj1, obj2, geo* and *pic*

txt-src a collection of selected textual and source code files from the Calgary Corpus with most of them repeating twice in the following order: *paper1, progc, paper2, progp, news, paper1, progc, paper2* and *progp*

kennedy.xls an MS Excel Spreadsheet from the Canterbury Corpus [16]

world192.txt a text file, The CIA world fact book, from the Large Corpus [17]

For most of these data samples, the improvement contributed by *Weighted History* is even more significant, and the WH enabled variant manages to outperform PPMII (also CTCH+II+SEE without WH closes the gap to PPMII for these data). Neither of this is true for the file *world192.txt*, which is better compressed by PPMII, and also the effect of WH is negligible for it (suggesting that the file is almost perfectly homogeneous).

It is interesting to see that the shuffled Calgary Corpus collection is compressed better by both CTCH variants than the sorted one (in contrast to PPMII). The *txt-src* file collection was intended to demonstrate benefits of WH, and the improvement is indeed significant.

It should also be noted that PPMII behaves strangely on the file *kennedy.xls*; its performance is poor with the maximum order 16, but it achieves considerably better result (1.02 bpB) when limited to order 8.

The results show that it is possible to benefit from considering relevance of individual occurrences (or relevance of locations where they appeared), but the approach also brings an inherent increase of the computational complexity; therefore, it is not a surprise that our implementation can't compete in terms of speed and memory requirements. CTCH+II+SEE needs about 100 seconds (most of it is consumed by the II approximation, though) and 220 MB of memory to compress *book1*; its WH enabled variant requires about 148 seconds and 228 MB of memory, which are extreme values compared to 0.3 s and 16 MB required by PPMII to compress the same file on the same computer (decompression times are very similar to respective compression times for all of these methods).

However, our implementation contains only very basic optimizations made to enable reasonable testing, and the time required by the WH enabled variant can be reduced to 61 seconds only by relaxing some parameters (blending only three top levels and using only the three thousand most recent occurrences of each context) resulting in a degradation of the compression efficiency by only 0.14 % on average for the files of the Calgary Corpus. Memory requirements can also be greatly reduced by using more efficient storage of the history indexes and the context statistics. Nevertheless, our implementation achieved the main goal—to verify that the concept of *Weighted History* is meaningful.

Chapter 5

Conclusion

We suggested two algorithms from different areas of the context modelling that share one common element—they both try to exploit the data history in an innovative way to address some known problems and to explore some new ideas.

In chapter 3, we described an algorithm for building the context tree which addresses the problem of inaccurate probabilities in high orders. In the literature [14, 9, 3], there is a consensus that the problem is caused by insufficient statistics (long contexts occur too rarely). Although we agree that it is a significant cause, we claim that it is not the only cause, and it might not even be the most important one.

We propose that comparably important cause of the problem is ignoring previous occurrences of a context before it is added into the context tree. We demonstrated this claim by two examples and discussed already known solutions addressing the cause. Then we identified some more requirements for an ideal context tree. The main part of the chapter presents an algorithm for maintaining the context tree that completely removes the problem of ignoring some context occurrences and also fulfills the additional requirements. The satisfaction of our requirements was supported by several propositions, which included informal proofs.

The empirical results confirmed that the algorithm improves compression efficiency at high context orders, and that it removes the unfortunate effect of degrading compression performance with growing maximum order. According to the results, an approximation of *Information Inheritance* can further intensify the trend of improving performance with the growing order. Unfortunately, in terms of absolute compression performance, our implementation did not meet our expectations; it performs worse than other feature-equivalent algorithms.

In chapter 4, we proposed a method that tries to estimate relevance of individual parts of the history (called sections) in a more elaborate way than just

favoring the most recent data. We suggested possible indicators of relevance of the data sections, and proposed a concept utilizing them. Then we simplified the concept to allow its implementation with a reasonable effort. One of the approximations was using long contexts instead of keywords to identify different nature of parts of data (e.g., topics in text). A description of our implementation followed; it reuses the same data structure as the algorithm for context tree maintenance, and its another important feature is the secondary model for estimating the section weights.

The empirical results showed that the method improved compression performance for almost all files of the Calgary Corpus (by 1.2% on average), and that the effect was even more significant on some larger data samples (it scored improvements around 3% on 5 of 6 tested samples); though, at the expense of very low compression speed.

The method is certainly not usable in practical applications as it is, but it tries to explore a new direction in the context modelling. For example, some elements of the method could be used for combination of different models for distinct types of data, or for selection of a convenient model for some specific data.

On the contrary, the algorithm for building the context tree presented in chapter 3 can be relatively easily adapted for use in today's practical algorithms which use the context tree, such as PPMII. Nevertheless, the algorithm brings an increased computational complexity, and will probably need a more optimized implementation (or even some approximations) before it is widely practically usable.

5.1 Further Work

We verified the main properties of the CTCH algorithm we needed, but its implementation has some significant drawbacks. First, it should be examined why it performs worse then PPMD at low orders. It can be done by careful comparison of probability distributions produced by the algorithms or by implementing the CTCH algorithm into the code-base of PPMD. Both options are not easy to do but certainly not unfeasible.

Further, CTCH could be implemented in a much more CPU and memory efficient way (and ideally run in a bounded memory). It might also be useful to examine how it behaves when constrained (for example, by limiting length of the history indexes during the context tree update) to allow balancing between speed and compression efficiency.

The *Weighted History* concept can be further explored in many possible directions; for example, some of the simplifications applied to the original concept can be revised to examine its potential.

Appendix A

Detailed Empirical Results

This appendix provides exact data for the graphs presented in section 3.7 starting on page 41.

Table A.1 shows compression performance of the four methods discussed in the above-mentioned section for different maximum lengths of contexts; it is an unweighted average on all 14 files of the Calgary Corpus [15].

Algorithm / Order	4	5	6	8	10	16	255
CTCH	2.391	2.356	2.345	2.338	2.336	2.333	**2.326**
CTCH + II	2.378	2.339	2.325	2.314	2.310	2.305	**2.288**
PPMD	2.310	**2.290**	2.297	2.319	2.339	2.369	
PPMD + II	2.254	2.211	2.197	2.186	2.184	**2.182**	

Table A.1: Source data for graph 3.8 on page 42 [bpB]

Table A.2 shows CTCH results for various types of data from the Calgary Corpus.

File Group / Order	4	5	6	8	10	16	255
Plain text	2.267	2.231	**2.230**	2.242	2.250	2.255	2.255
Binary objects	2.117	2.092	2.082	2.068	2.063	2.057	**2.055**
Source codes	2.014	1.970	1.941	1.912	1.899	1.884	**1.877**
Semi-structured	2.007	1.951	1.930	1.915	1.908	1.904	**1.901**

Table A.2: Source data for graph 3.9 on page 43 [bpB]

Appendix B

Enclosed CD Contents

The enclosed CD contains the following items:

thesis/ContextModelThesis.pdf this document

sources/ source codes of the reference implementation

doc/ programmer's documentation in the HTML format (generated by Doxygen from the sources)

bin/ executable binaries (Linux 32-bit) built from the sources, the following variants:

- **ctch** the basic variant presented in chapter 3 (CTCH)
- **ctch-ii-see** the variant with *Information Inheritance* approximation and *Secondary Escape Estimation* enabled (CTCH+II+SEE) used as a comparison baseline in chapter 4
- **ctch-ii-see-wh** the *Weighted History* enabled variant (CTCH+II+SEE+WH) presented in the same chapter

Appendix C

Compilation and Usage of the Implementation

The reference CTCH implementation consists of a single program that allows to enable or disable *Information Inheritance* approximation, *Secondary Escape Estimation* and *Weighted History* at compile-time.

C.1 Compilation

The implementation is written in C++ (dialect ANSI/IEC 98). It uses the unit testing framework UnitTest++ [19] (and the standard C++ library), and it is built with the help of GNU Make (recommended version 3.81). The recommended compiler is the C++ one from GNU Compiler Collection, version 4.1.2. A GNU/Linux environment is assumed (although porting to MS Windows should be possible with a small effort).

To produce an executable binary, simply running make in the directory with sources is sufficient, but before doing that, change of some compile-time parameters to enable or disable some specific features may be required. The parameters are contained in the following source files and are described in the programmer's documentation:

WHModel.h allows to enable or disable the main features (*Weighted History*, SEE, II approximation) and configure many other parameters such as maximum context order

HistoryModel.h allows detailed configuration of *Weighted History* (when it is enabled)

SEE.h contains all parameters of *Secondary Escape Estimation*

Makefile allows enabling of debugging support or using an alternative compiler (that needs to be set up also in tests/Makefile and tests/unittest-cpp/Makefile)

C.2 Usage

The program name is "3C" (stands for Calgary Corpus Challenger). Its first argument is always name of the archive; if the archive file is present, 3C tries to decompress it; if not, one or more files to be put into the archive must be specified as arguments following the archive name. When decoding, the original files are overwritten if they are present.

The following example demonstrates creating an archive:

```
$ ./3C sources.3c progc progl progp
 Uncompressed size: 160636 bytes
 File(s) processed in 8.150 seconds at 19.25 kB/s.
```

To decompress the archive, the following command must be run:

```
$ ./3C sources.3c
 Uncompressed size: 160636 bytes
 File(s) processed in 10.270 seconds at 15.27 kB/s.
```

Bibliography

[1] Moffat, A., Neal, R. and Witten, I. H. (1990) *Arithmetic Coding Revisited.* Proc. Data Compression Conf. IEEE Computer Society Press, Los Alamitos, CA.

[2] Bunton, S. (1996) *On-Line Stochastic Processes in Data Compression.* Ph.D. Thesis, University of Washington, Seattle, WA.

[3] Cleary, J. G. and Witten, I. H. (1984) *Data Compression Using Adaptive Coding and Partial String Matching.* IEEE Trans. Commun., COM-32, 396-402.

[4] Witten, I. H. and Bell, T. C. (1991) *The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression.* IEEE Trans. Inform. Theory, IT-37, 1085-1094.

[5] Moffat, A. (1990) *Implementing the PPM Data Compression Scheme.* IEEE Trans. Commun. COM-38, 1917-1921.

[6] Howard, P. G. (1993) *The Design and Analysis of Efficient Lossless Data Compression Systems.* Ph.D. Thesis, Brown University.

[7] Cleary, J. G., Teahan, W. J. and Witten, I. H. (1995) *Unbounded Length Contexts for PPM.* Proc. Data Compression Conf., March, 52-61.

[8] Bloom, C. (1998) *Solving the Problems of Context Modeling.* www.cbloom.com/papers/

[9] Shkarin, D. (2002) *PPM: One Step to Practicality.* Proceedings of the IEEE Data Compression Conference (DCC'2002), 202-211.

[10] *ppmd v9.1-14;* Based on author's original PPMII implementation. http://http.us.debian.org/debian/pool/main/p/ppmd/ (available also as a package in common GNU/Linux distributions)

[11] Mahoney, M. (2002) *The PAQ1 Data Compression Program.* http://www.cs.fit.edu/~mmahoney/compression/paq1.pdf

[12] Mahoney, M. (2005) *Adaptive Weighing of Context Models for Lossless Data Compression.* Florida Tech. Technical Report CS-2005-16, http://www.cs.fit.edu/~mmahoney/compression/cs200516.pdf

[13] *The PAQ Project.* http://www.cs.fit.edu/~mmahoney/compression/#paq

[14] Moffat, A. and Turpin, A. (2002) *Compression and Coding Algorithms.* Kluwer Academic Publishers

[15] *The Calgary Corpus.* http://corpus.canterbury.ac.nz/descriptions/#calgary

[16] *The Canterbury Corpus.* http://corpus.canterbury.ac.nz/descriptions/#cantrbry

[17] *The Large Corpus.* http://corpus.canterbury.ac.nz/descriptions/#large

[18] *The Hutter Prize.* http://prize.hutter1.net/

[19] *UnitTest++* http://unittest-cpp.sourceforge.net/

[20] *GNU General Public License.* http://www.gnu.org/licenses/gpl.html