PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Implementación de un algoritmo de codificación universal eficiente usando árboles de contexto

JORGE MERLINO

2014

Jorge Merlino
Implementación de un algoritmo de
Codificación universal eficiente usando
arboles de contexto.
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 14-05
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, 2014

Implementación de un algoritmo de codificación universal eficiente usando árboles de contexto

Tesis de maestría

Jorge Merlino

jmerlino@fing.edu.uy

Palabras clave: Compresión, Codificación aritmética, Modelos árbol, Árboles de contexto, Codificación semi-predictiva

Tutor: Alfredo Viola viola@fing.edu.uy

Facultad de Ingeniería
Montevideo, Uruguay

9 de diciembre de 2013

Resumen

En este trabajo se propone una nueva variante del algoritmo *context* [28] usando codificación semi-predictiva y se desarrolla una implementación eficiente del mismo.

Para manejar los problemas relacionados con la gran cantidad de parámetros típicos de los archivos de texto y ejecutables y mejorar el nivel de compresión del algoritmo, se adaptaron algunas técnicas principalmente de algoritmos tipo *PPM* [8]. El procedimiento de podado y el estimador de probabilidad fueron adaptados para tener en cuenta estos cambios.

Además se usó una versión modificada del algoritmo *wotd* [11] para podar el árbol durante su construcción reduciendo así los requerimientos de memoria durante la fase de codificación del algoritmo.

Finalmente se realizó una comparación experimental entre el algoritmo desarrollado aquí y algunas de las mejores implementaciones de otros algoritmos de codificación conocidos.

Efficient Universal Coding Algorithm Implementation Using Context Trees

Master thesis

Jorge Merlino

jmerlino@fing.edu.uy

Keywords: Compression, Arithmetic coding, Tree models, Context trees, Semi-predictive coding

Advisor: Alfredo Viola viola@fing.edu.uy

Facultad de Ingeniería
Montevideo, Uruguay

December 9, 2013

Contents

List of Algorithms

Abstract

In this work, a new *context* algorithm [28] using the semi-predictive approach is proposed and an efficient implementation of the algorithm is developed.

In order to deal with issues related to the large number of parameters typical of text and executable files and improve the compression rate of the algorithm, some techniques, mainly from *PPM*-like algorithms [8], were adopted. The prunning procedure and probability estimator were adapted to account for these modifications.

Also, a modified version of the *wotd* algorithm [11] is used to prune the tree during its construction reducing the memory footprint of the encoding phase of the algorithm.

Finally, an experimental comparison was performed between the algorithm developed here and some of the best implementations known of other compression algorithms.

Chapter 1

Introduction

Data compression or source coding is the process of representing information using fewer bits than would be needed by an uncompressed representation. Lossless compression algorithms exploit the statistical redundancy in the data to find a more concise representation without losing information.

The data to compress is usually modeled as coming from a data source which generates symbols from a countable (usually finite) alphabet. As part of the compression process a statistical model will be developed and assigned to the data source. This model will be derived from the data with the objective of assigning probabilities to the symbols generated by the source. The symbols that occur more frequently should have greater probabilities than those that occur less. It has been proved [34] that the entropy (see 1.3) of the probability distribution of the symbols imposes a fundamental limit in the code length per symbol that can be achieved for a given data source.

In this context, the research and development of compression algorithms with proven theoretical and practical properties is important. On the theoretical side the key is to have algorithms whose compression rate is bounded by known formulas and that this compression rate converges asymptotically to the entropy of the source. There are codes that achieve this and there are others that have optimal compression rate bounds not only for a single probability model but for a whole class of them. These are called *universal codes* (see 1.6) and they can asymptotically achieve a compression rate as good as any code especially designed for a particular model in the class. The rate of convergence to the entropy is also important as, within universal codes, there is a proved bound by Rissanen for the rate of this convergence for certain models which depends on the number of free parameters of the model [30].

On the practical side the fundamental trade-off is established between processing speed, resource consumption and compression ratio. A practical algorithm should be capable of compressing large files with a modest consumption of memory and time and, at the same time, obtain reasonable compressed file sizes.

There is a plethora of compression algorithms available but only a few of them have both proved theoretical properties and good practical performance. The work developed here is based on the linear time context algorithm presented in [20]. This algorithm is proved to have linear time complexity and is also optimal in a double universality framework (see 1.8) with respect to the already mentioned Rissanen's lower bound [30].

We survey some of the best known compression algorithms and their implementations. The four different paradigms studied are CTW [39, 40], BWT [7], PPM [8] and finally the class of context mixing algorithms [18]. We survey both the theoretical and practical properties of each one using the most efficient implementations that have been proposed.

The first three of these algorithms and also the context algorithm, are based on context models which use a number of the previously emitted symbols by the source as the basis for the estimation of the probability of the next one. These are in fact Markov models (see 1.7.2) of order k where k is the number of the previously emitted symbols considered by the model. Except for BWT they all use tree models (see 1.7.3) to represent contexts of variable length on different locations. Tree models reduce drastically the number of free parameters as compared with a Markov model for the same source.

We also present a new algorithm based on the one in [20] and an implementation of this new algorithm. The compression rate of our implementation was second best below PPMd [35] in most

tests. One key property of this implementation is the use of the *wotd* algorithm presented in [11] to prune the context tree during its construction instead of building the complete tree and prune it in a second phase as was suggested in the original algorithm. This reduces the memory needs of the compression process (although it does not have linear execution time) and in practice reduces the total time needed to compress most files.

Another improvement of the new algorithm is the use of several techniques found in other PPM based algorithms. These include the possibility of coding a symbol in a different state than the one originally selected by the algorithm if the symbol to code has not occurred before in that state instead of assigning the same probability to all unknown symbols. Some additional modifications to the algorithm were needed to enable this, including the definition of a new cost function and a new recursive procedure for pruning the tree that takes into account this new behavior.

Finally we studied the PPMd algorithm presented in [35] and give a complete description of the algorithm tree construction and updating. This information was not present in the literature.

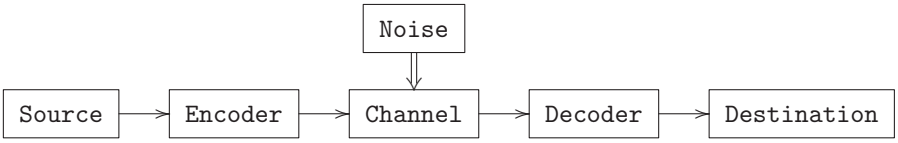
The remainder of this thesis is organized as follows. The rest of this section introduces some fundamental mathematical and algorithmic concepts used throughout this paper. Next, chapter 2 describes the Context Tree Weighting (CTW) compression algorithm and the particular implementation of this algorithm used for comparison in this work. In a similar way chapter 3 describes the Prediction by Partial Matching (PPM) algorithm and the implementation surveyed in this work. Furthermore, in chapter 4, section 4.1 introduces the Burrows-Wheeler Transformation (BWT) and a popular compression algorithm (Bzip) that uses this transformation that was also used for comparison and section 4.2 describes the class of context mixing algorithms. Finally, chapter 5 defines the Context compression algorithm and describes the new implementation of this algorithm and chapter 6 presents experimental analysis of the behavior of this implementation and comparisons to the previously described algorithms.

1.1 Notation

Let A be an alphabet of $\alpha \geq 2$ symbols and let λ denote the empty string. Let also the string $a_1, a_2 \dots a_n,\ a_i \in A$ be denoted by a_1^n. More generally, the notation a_j^k is used as a shorthand for $a_j, a_{j+1} \dots a_k,\ a_i \in A,\ j \leq i \leq k$. Also the subscript is omitted when $j = 1$ i.e., $a^k = a_1^k$. For $a = a^k$ we let $|a| = k$ denote the length of a, $\bar{a} = a_k a_{k-1} \dots a_1$ its reverse string and $head(a)$ its first symbol a_1 or λ if $k = 0$. For strings u, v the concatenation of u and v is denoted as uv.

1.2 Communication systems and coding

Information theory is concerned with the study of communication systems which are generally represented by the following diagram:



The source generates data that need to be communicated to the destination. The encoder transforms data into messages suitable for transmission over the channel. The channel is the medium over which the coded message is transmitted; it can be affected by noise that alters the transmitted message. Then, the decoder receives the output of the channel and tries to recover the original message for delivery to the destination.

The encoder can be seen as two different processes: one that removes redundancy from the message in order to create a smaller message to be transmitted through the channel (*source encoder*) and another one that adds redundancy to the message with the purpose of making the message resilient to alterations caused by noise in the transmission channel (*channel encoder*). The decoder is symmetrical and has a *channel decoder* to remove the redundancy and possibly correct the errors introduced by the noise in the channel and a *source decoder* that retrieves the original message.

This work will be focused on source coding and decoding. The channel of communication and the possible noise added therein will not be considered.

1.3 Entropy

The entropy of a random variable is defined by Shannon in [34] as a measure of the uncertainty of such variable. Let X be a discrete random variable with alphabet A and probability mass function $p(a) = Pr\{X = a\}, a \in A$. The entropy $H(X)$ of X is defined as:

$$H(X) = -\sum_{a \in A} p(a) \log p(a).$$

The entropy of X can also be interpreted as the expected value of the random variable $\log \frac{1}{p(X)}$:

$$H(X) = E\left(\log \frac{1}{p(X)}\right).$$

The entropy is considered as measuring the self-information of X based on this interpretation.

As shown before, to calculate the entropy we need a probability distribution. When we have a given string x^n and the true distribution of the source that generated that particular sequence is unknown we define an empirical distribution of the symbols of x^n and calculate the entropy using this distribution. The empirical probability for a symbol a in a particular sequence x^n is defined as:

$$\hat{p}_{x^n}(a) = \frac{1}{n} \sum_{i=1}^{n} 1_a(x_i),\ a \in A,$$

where 1_a is the indicator function for $\{a\}$ (the set whose only member is a). The indicator function for a set A and an iid model is defined as:

$$1_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}.$$

In this case the probabilities are estimated using the data in x^n and assuming that the process that generated x^n is ergodic and stationary (this implies asymptotic convergence of the empirical distribution to the distribution of the source). When the entropy is calculated over an empirical distribution it is called the *empirical entropy* of the string.

1.4 Source codes

A source code C for a random variable X with alphabet A is a mapping $C : A \to D^*$ where D is a finite coding alphabet and D^* is the set of finite strings over D.

We will also define $C(a)$ as the codeword corresponding to a, $a \in A$ and $l(a) = |C(a)|$ will be the length of such codeword. Thus, the *expected length* of $C(X)$ for $X \sim p(a)$ is defined as:

$$L(C) = E_p[l(a)] = \sum_{a \in A} p(a) l(a). \tag{1.1}$$

When coding strings of symbols we can do it symbol by symbol and define the *extension* C^* of a code C as the mapping from A^* to D^*. C^* is defined as the concatenation of the codewords for each symbol of the string i.e:

$$C^*(x_1 x_2 \dots x_n) = C(x_1) C(x_2) \dots C(x_n).$$

We want to be able to decode each symbol of C^* without reference to future codewords so we require that no codeword of C is a prefix of any other codeword. A code that achieves this property is called *prefix free* or *instantaneous* code.

It has been proved by Shannon in [34] that any prefix free code for a set of symbols with distribution $X \sim p(a)$ has $H(X)$ as a lower bound on its expected code length and that we can build a code that satisfies

$$H(X) \leq L(C) \leq H(X) + 1.$$

It can be easily deduced from (1.1) (assuming a finite alphabet and a finite string) that a symbol which occurs with probability $p(a)$ is best represented by a code where $l(a) = -\log p(a)$. This ideal code

length cannot be achieved in general because $-\log p(a)$ is not necessarily an integer. The challenge is to build practical codes that achieve a number of bits per symbol as close to the entropy as possible.

When dealing with a sequence of symbols, instead of using the extension code C^* we can use an *alphabet extension*

$$A^n = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A\}$$

and define a new code C^n for blocks of symbols $X^n = (X_1, X_2, \ldots, X_n)$. It is also shown in [34] that a code exists that satisfies

$$\frac{H(X^n)}{n} \leq \frac{L(C^n)}{n} \leq \frac{H(X^n)}{n} + \frac{1}{n}$$

which is better per symbol than what can be achieved by a symbol by symbol extension code.

A related notion is the so called *pointwise redundancy* of a code that is defined as the difference between the codeword length for a string a^n and the ideal code length given by $-\log p(a^n)$. Thus we define the pointwise redundancy for a^n given a code C and probability distribution p assigned by a model for the data source as:

$$R_{C,p}(a^n) = l(a^n) + \log p(a^n).$$

We can also define the *expected redundancy* as the expectation of $R_{C,p}(X^n)$ with respect to p as:

$$\overline{R}_{C,p} = E_p[R_{C,p}(X^n)] = L(C) - H(X^n).$$

In practice, when we are encoding a string of symbols we are not given a probability distribution of such symbols. To be able to exploit the statistical regularities of the data in order to compress the string it is assumed that the symbols are generated by a process that follows a probability model. There are different types of models that can be used, and depending on the model some parameters have to be determined to fit the model to the data to compress. These estimated parameters should maximize the probability of the string according to the model in order to achieve the shortest code length.

When we are coding a string of symbols of length n we need the model to assign a probability to each of the a^n possible strings. Since a^n is sequentially read the empirical probability distribution of the symbols read up to a^j change when reading the symbol a_{j+1}. Thus we need to assign probabilities sequentially like:

$$\begin{cases} Pr(\lambda) & = 1 \\ Pr(ua) & = Pr(u)Pr(a|u), a \in A, u \in A^* \end{cases}. \tag{1.2}$$

This definition implies a sequential algorithm to assign probabilities. As a consequence we need efficient ways to adaptively change this empirical distribution while scanning a^n. The final distribution that has been estimated for a^n within the model will be used for compressing it. This is called *adaptive coding* [3].

1.5 Arithmetic coding

Arithmetic coding allows the computation of a codeword for a source sequence in an efficient and sequential way with an expected code length very close to the ideal ($H(X)$ bits per symbol) up to an additive constant. Arithmetic codes are based on the Elias algorithm (unpublished but described in [1, 13]) but only became feasible after Rissanen [27] and Pasco [26] solved the issues of arithmetical precision involved in the initial algorithm.

The Elias algorithm (also known as Shannon-Fano-Elias code) is based on the notion of the cumulative distribution function. Let us suppose the alphabet is ordered (for example in lexicographical order). Then, the cumulative distribution function for the symbol a_i is defined as:

$$F(a_i) = \sum_{x<a_i} p(x)$$

that is, the sum of the probabilities of all symbols less that a_i in this order. A variation of this formula is also used in practice:

$$\bar{F}(a_i) = \sum_{x<a_i} p(x) + \frac{1}{2}p(a_i).$$

Assuming that the probabilities of all symbols are positive (symbols with probability 0 could be excluded) $\bar{F}(a_i)$ is a strictly increasing function which represents the middle point between $F(a_i)$ and $F(a_{i+1})$. Then, a_i can be uniquely determined if we know $\bar{F}(a_i)$. Thus the value $\bar{F}(a_i)$ can be used as a code for a_i.

The code $\bar{F}(a_i)$ is a real number between 0 and 1. $\bar{F}(a)$ is always less than unity because the maximum value of $\bar{F}(a)$ is $\bar{F}(a_\alpha)$ and:

$$1 = \sum_{x \in \{a_1, \ldots, a_\alpha\}} p(x) = \sum_{x < a_\alpha} p(x) + p(a_\alpha) > \bar{F}(a_\alpha).$$

The decimal part of $\bar{F}(a_i)$ can be expressed in binary notation as a binary string.

In general $\bar{F}(a_i)$ could be a real number only expressible by an infinite number of binary digits, so the value is truncated to $\lceil -\log p(a_i) \rceil + 1$ digits (bits). Let us represent this truncation for number y as $\lfloor y \rfloor$. It can be shown that the value of $\lfloor \bar{F}(a_i) \rfloor$ is still between $F(a_i)$ and $F(a_{i+1})$. To see this, first it has to be noticed that the difference between $\bar{F}(a_i)$ and $\lfloor \bar{F}(a_i) \rfloor$ is less than $\frac{1}{2^{\lceil -\log p(a_i) \rceil + 1}}$, so we have:

$$\frac{1}{2^{\lceil -\log p(a_i) \rceil + 1}} < \frac{1}{2^{-\log p(a_i)} 2} = \frac{p(a_i)}{2} = \bar{F}(a_i) - F(a_i)$$

so $\lfloor \bar{F}(a_i) \rfloor$ is an adequate code for a_i.

We next need to show that this code is prefix free. If we consider an unlimited number of bits the codewords will eventually be prefix free because the real numbers representing $\bar{F}(a_i)$ are all different. However, we need to show that the truncation to $\lceil -\log p(a_i) \rceil + 1$ preserves this property. To prove this we can consider that, after the truncation, each codeword represents an interval of length $\frac{1}{2^{\lceil -\log p(a_i) \rceil + 1}}$ which contains the middle point of the interval $[F(a_i), F(a_{i+1})]$. This interval I is defined such as:

$$\lfloor x \rfloor = \lfloor y \rfloor \; \forall \; x, y \in I.$$

This is to say that all numbers within this interval have the same truncated value. By the last result we know this codeword interval is fully contained in the $[F(a_i), F(a_{i+1})]$ interval so all the codeword intervals are disjoint and the code is prefix free considering the first $\lceil -\log p(a_i) \rceil + 1$ bits. Notice that all digits are part of the code even the last non-significant zeroes.

Since we are using a code with length $\lceil -\log p(a) \rceil + 1$ to represent a the expected code length of this code is:

$$\begin{aligned} L(C) &= \sum_{a \in A} p(a) l(a) \\ &= \sum_{a \in A} p(a) \left(\lceil -\log p(a) \rceil + 1 \right) \\ &< \sum_{a \in A} p(a) \left(-\log p(a) + 2 \right) \\ &= \sum_{a \in A} (p(a) - \log p(a)) + \sum_{a \in A} 2\, p(a) \\ &= H(X) + 2 \end{aligned}$$

so this is within two bits of entropy (the ideal).

This method can be used to code strings of symbols using the cumulative distribution for the sequence defined as:

$$F(a^n) = \sum_{x^n < a^n} P(x^n)$$

where we assume an order on the strings based on the order of the individual symbols. In this case we can code a string using a point in the particular interval (one of the α^n intervals) that corresponds to the string being coded. This interval, for a string a^n, is defined as:

$$I(a^n) = \left[F(a^n),\; F(a^n) + P(a^n) \right).$$

The problem is that, as n gets larger, we need to calculate and add an exponential number of probabilities with very precise arithmetic as the intervals for each string are incrementally smaller. This problem is solved by arithmetic coding.

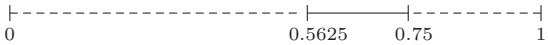
Before describing the algorithm for arithmetic coding it is important to remark that when coding with intervals as shown before it is possible to compute these intervals sequentially transforming the initial interval $[0, 1)$ into a series of nested intervals $I(a_1), I(a_2), \ldots, I(a_n)$ using the following chain of identities:

$$\begin{aligned} F(a^n) &= \sum_{x^n < a^n} P(x^n) \\ &= \sum_{x^{n-1} < a^{n-1}} P(x^{n-1}) + \sum_{x < a_n} P(a^{n-1}x) \\ &= F(a^{n-1}) + P(a^{n-1}) \sum_{x < a_n} P(x \,|\, a^{n-1}). \end{aligned} \tag{1.3}$$

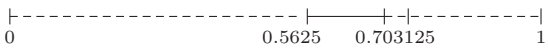
Therefore the encoder and the decoder can sequentially find $I(a^n)$ after having determined $I(a^{n-1})$ if it is easy to determine the conditional probabilities $P(x|a^{n-1})$. As mentioned before a model of the data is needed to obtain these probabilities.

For example lets define a static binary model with $P(0) = 0.75$ and $P(1) = 0.25$. A symbol 0 can be represented by any number in the interval $[0, 0.75]$.

If then a 1 is received the interval will now be $[0.5625, 0.75]$ (one fourth of the previous interval).



A following 0 will then result in $[0.5625, 0.703125]$



The probability for this string of three symbols is:

$$P(010) = \left(\frac{3}{4}\right)^2 \frac{1}{4} = 0.140625$$

and the ideal code length will be:

$$\lceil -\log p(010) \rceil = \lceil -\log 0.140625 \rceil = 3 \text{ bits}$$

which could be achieved by the binary number 0.101 which equals to the number 0.625 in base 10 which falls into the final interval. In this simple example the code has the same length as the original string but as more symbols are processed the code will be shorter than the input string.

Next we show the pseudo-code for the arithmetic encoder (algorithm 1). The algorithm uses two values *low* and *high* to keep track of the bounds of the current interval and needs to receive $F(a_i)$ and $F(a_{i-1})$ for every symbol to encode. These values will change for every symbol if some kind of adaptive encoding is used.

During the decoding of the message, the decoder will receive the value determined by the encoder and then, based on the $F(a_i)$ and $F(a_{i-1})$ will be able to obtain the encoded symbol. After finding out the symbol the interval can be rescaled using the same operations used by the encoder and then a new symbol can be decoded. If adaptive coding is being used the past decoded symbol will update the values of $F(a_i)$ $\forall a_i \in A$ so that these values are synchronized with the ones used by the encoder for the next symbol. See algorithm 2.

In practice this algorithm could require infinite precision computations as the interval is monotonically decreasing in size. This problem can be solved by noticing that, as the left bound and right bound of the interval get closer their first bits start to coincide [26, 27]. For example, looking at the last interval from the past example $[0.5625, 0.703125]$ if we express it in binary we get: $[0.1001, 0.101101]$. We can see that the first two bits of the interval's limits are the same so we are sure that all the numbers in the final interval will start with 0.10.

Algorithm 1: Arithmetic encoder

input : String x^n to compress
output: A point in the last interval

1 $low = 0$
2 $high = 1$
3 **for** $x_i = x_1$ **to** x_n **do**
4 $\quad range = high - low$
5 $\quad j =$ `SymbolIndex` (x_i)
6 $\quad low = low + (range\, F(a_{j-1}))$
7 $\quad high = low + (range\, F(a_j))$
8 **end**
9 **return** A point in the interval $[low, high)$

Algorithm 2: Arithmetic decoder

input : Encoded number $value$
output: Each symbol a_i

1 $low = 0$
2 $high = 1$
3 **for** $i = 1$ **to** n **do**
4 $\quad$ Find symbol a_j so that $F(a_j) >= (value - low)/(high - low) > F(a_{j-1})$
5 $\quad$ **return** a_j
6 $\quad range = high - low$
7 $\quad low = low + (range\, F(a_{j-1}))$
8 $\quad high = low + (range\, F(a_j))$
9 **end**

In this case, the encoder keeps a buffer for each of the two boundaries and, instead of generating a point in the interval for each encoded symbol, just outputs bits as they are removed from both buffers when they coincide. Usually 16 bits are enough for this buffer unless there are many symbols or some symbols with very low probability. In case these assumptions are false the product in lines lines 6 and 7 of algorithm 1 can be less than 2^{-16} and thus the interval would become empty as both boundaries will be equal. As will be shown in section 5.3 our algorithm halves the number of occurrences of symbols when they reach a determined threshold, this prevents that symbols reach very low probabilities and thus a 16 bit arithmetic coder can be used.

When the first bits in both buffers are the same these bits can be outputted by the encoder and the intervals can be shifted and rescaled so that the computations can be achieved with finite precision. Continuing with the example we can output the two known bits and rescale the interval into $[0.01, 0.1101]$ which in decimal is $[0.25, 0.8125]$

There is a possible problem when the interval's bounds are very close but their first bits do not agree. For example the interval in binary could be something like this: $[0.0111111, 0.10000000]$. Here the interval is very small but no digits can be outputted. If this situation continues we might need arbitrarily precise arithmetic to calculate the interval boundaries. This is called the *carryover* problem [27].

This situation can be avoided if the algorithm checks when the most significant digits of the two bounds do not match and the second most significant digit is one on the left bound and zero on the right bound (ie. if the interval looks like $[0.0\mathbf{1}\ldots, 0.1\mathbf{0}\ldots]$). In this case the two second most significant bits are deleted (the bold digits in the example) and the most significant bits are left untouched. As these two deleted bits are lost a count is taken of the bits that where deleted on account of this carryover problem.

The algorithm carries on in this way by processing more symbols and increasing the count in case this problem persists and more symbols are removed. When the two interval's bounds first digits agree, depending on the value of the most significant bit, it is decided that all of the deleted carryover bits are now zeroes or ones and then they are all outputted together and the count of deleted bits is

set to zero. If the most significant bit of the two bounds is one when the problem is solved all the carryover bits are determined to be zero and they are one otherwise.

Continuing with the example in the interval $[0.0\mathbf{1}\ldots, 0.1\mathbf{0}\ldots]$ suppose we remove the two bold bits and after processing some more symbols we end up with an interval looking like $[0.10\ldots, 0.11\ldots]$. We then know that the value the arithmetic encoder will output in fact starts with 0.1, so the value of the numbers we removed equals to the value they had in the right bound of the interval. Including these numbers the interval should be $[0.1\mathbf{0}0\ldots, 0.1\mathbf{0}1\ldots]$. If, on the other hand, the final interval is like $[0.00\ldots, 0.01\ldots]$ then the real interval should be $[0.0\mathbf{1}0\ldots, 0.0\mathbf{1}1\ldots]$. In either case the first two symbols can be sent to the output and the interval can be rescaled.

The decoder keeps an extra buffer besides the *low* and *high* ones that saves the bits received from the encoder (it represents the *value* in algorithm 2). It starts working as soon as it can fill this buffer and then, as its intervals are synchronized with the encoder's, it can infer when more symbols need to be shifted in as the most significant digits are shifted out from the buffer.

It can be shown ([26]) that the code length for a sequence a^n assigned by a properly designed arithmetic coder satisfies:

$$l(a^n) < \log \frac{1}{P(a^n)} + 2$$

so the coder is optimal up to a two bits redundancy.

There is a variant of this algorithm in the form of the so called *Range coders* [22]. The theoretical ideas are the same but range coders can use intervals larger than $[0, 1)$. They define a scale s that determines an interval $[0, s)$ and divide this interval completely in subintervals the same way as the arithmetic coders do.

1.6 Universality

As mentioned before, in order to perform a lossless compression of a sequence of symbols a proper model of the data is needed. The model is used to assign probabilities to be used by an arithmetic coder. This can be a problem in many practical applications where a model of the data is not known in advance or we only have an individual sequence to be compressed with no probability distribution underlying the data at hand. In these cases it is interesting to find out how much a sequence can be compressed. To find this out it is necessary to impose a restriction on the class of possible models as we can always find a probability model precisely fitted for a particular sequence that assigns probability one to this sequence compressing it to one bit while leaving every other sequence uncompressed.

We can define a model class from a particular model leaving some free parameters that can be adjusted in order to fit the model to some particular sequence. The class of models would be formed by all the possible models that can be instantiated by assigning all possible values to the model class' free parameters. For example, we could define a model for a binary alphabet that assigns probabilities using a Bernoulli distribution with parameter θ. This is a *memoryless* model because the probability of each symbol does not depend on the previous symbols. In this case the class of models would be the set of all possible Bernoulli models for any value of the parameter θ.

It is in this setting that we define a *universal code* for some class $\mathcal{C}$ as one that performs asymptotically as well as any model in the class $\mathcal{C}$ for any string. More formally, we define an indexed class of models $\{P_\theta, \theta \in \Lambda\}$ where Λ is the index set and require that the normalized expected redundancy for a sequence of n symbols vanishes for each θ:

$$\frac{1}{n}\overline{R}_{\mathcal{C},P_\theta}(a^n) \to 0 \text{ when } n \to \infty \quad \forall\, \theta \in \Lambda.$$

Moreover, the code is pointwise universal if:

$$\frac{1}{n} \max_{a^n \in A^n} R_{\mathcal{C},P_\theta}(a^n) \to 0 \text{ when } n \to \infty \quad \forall\, \theta \in \Lambda.$$

Choosing the model class is a very important issue because, even when having a universal encoder for a class, the larger the class the slower the convergence rate of the universal code to the best model in the class. There is a known bound by Rissanen [30] for the expected normalized redundancy for a block of n symbols that, under certain conditions, depends on the number of free parameters of the model (*model cost*). For a model with k free parameters its higher order term is $\frac{k \log n}{2n}$. It is

intuitive to think of this bound as meaning that previously known information about the process that generates the symbols helps to reduce the expected code length by reducing the number of free parameters of the model.

There are two common methods for probability assignment in universal codes: the mixture approach and the plug-in approach [24].

Plug-in codes estimate the index θ at every time instant t based on the previous a^{t-1} symbols and use the model specified by this index to assign probabilities as if it was the true parameter. For example the maximum likelihood estimator could be used at each instant. This estimator selects the value of the model parameter that produce a distribution that gives the observed data the greatest probability. In this case this is equivalent to using the empirical distribution of the symbols in a^{t-1} .

On the other hand mixture codes are based on generating convex combinations (mixtures) of all the models in the class. The coefficients of the convex combination (that must sum up to 1) can be seen as a prior probability on Λ

As an example let us consider the already discussed class of Bernoulli models defined as:

$$\{P_\theta, \theta \in \Lambda = [0,1]\}$$

where for each model in the class $p(1) = \theta$ and $p(0) = 1 - \theta$.

In this case, for a sequence a^n with n_0 zeros and n_1 ones we have:

$$\min_{\theta\in\Lambda} -\log P_\theta(a^n) = -\log P_{\hat{\theta}(a^n)}(a^n)$$

where $\hat{\theta}(a^n)$ is the maximum likelihood estimator for θ which in this case is the empirical distribution:

$$\begin{aligned} P_{\hat{\theta}(a^n)}(a^n) &= \hat{p}(0)^{n_0}\ \hat{p}(1)^{n_1} \\ &= \left(\frac{n_0}{n}\right)^{n_0}\ \left(\frac{n_1}{n}\right)^{n_1} \\ &= \frac{n_0^{n_0} n_1^{n_1}}{n^n}. \end{aligned}$$

We are then looking for a code with length function $l(a^n)$ such that:

$$\frac{1}{n}\left[l(a^n) + \log\left(\frac{n_0^{n_0} n_1^{n_1}}{n^n}\right)\right] \to 0 \text{ when } n \to \infty. \tag{1.4}$$

An example of such a universal code would be to enumerate all $\binom{n}{n_1}$ sequences with n_1 ones and describe the sequence with its index and also encode the value of n_1 resulting in a code length of:

$$l(a^n) = \left\lceil \log \binom{n}{n_1} \right\rceil + \lceil \log(n+1) \rceil.$$

Next, to show that this code satisfies (1.4) we have these inequalities:

$$\begin{aligned} \left\lceil \log \binom{n}{n_1} \right\rceil + \lceil \log(n+1) \rceil &\leq \log \binom{n}{n_1} + \log(n+1) + 2 \\ &= \log\left(\frac{n!}{n_1! n_0!}\right) + \log(n+1) + 2 \\ &= -\log\left(\frac{n_1! n_0!}{n!}\right) + \log(n+1) + 2 \\ &\leq -\log\left(\frac{n_1^{n_1} n_0^{n_0}}{n^n}\right) + \log(n+1) + 2. \end{aligned} \tag{1.5}$$

So, substituting this last expression for $l(a^n)$ in (1.4) we have:

$$\frac{\log(n+1)+2}{n} \to 0 \text{ when } n \to \infty$$

which trivially holds.

In (1.5) we are using Stirling approximation as:

$$n! > \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$
$$n! < \sqrt{2}\left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

so we then have:

$$\frac{n_1!n_0!}{n!} > \frac{\left(\frac{n_1}{e}\right)^{n_1}\sqrt{2\pi n_1}\left(\frac{n_0}{e}\right)^{n_0}\sqrt{2\pi n_0}}{\sqrt{2}\left(\frac{n}{e}\right)^n\sqrt{2\pi n}}$$
$$> \frac{n_1^{n_1}n_0^{n_0}}{n^n}\sqrt{\frac{\pi n_1 n_0}{n}}$$

and as

$$\frac{\pi n_1 n_0}{n} > 0$$

with $n_1, n_0 > 0$ (if $n_i = 0$ then $P_{\hat{\theta}(a^n)}(a^n) = 1$ and all this is trivial) we can conclude that:

$$\frac{n_1!n_0!}{n!} > \frac{n_1^{n_1}n_0^{n_0}}{n^n}.$$

Another example in the same class of Bernoulli models using the mixture approach would be to sum the values of all the models in the class assigning the same weight to each of them (*uniform mixture*). In a uniform mixture of θ:

$$\begin{aligned}\int_0^1 P_\theta(a^n)d\theta &= \int_0^1 \theta^{n_1}(1-\theta)^{n_0}d\theta \\ &= \frac{n_1!n_0!}{(n+1)!} \quad \text{(beta function integral)} \\ &= \frac{n_1!n_0!}{n!}\frac{1}{n+1} \\ &= \binom{n}{n_1}^{-1}\frac{1}{n+1}\end{aligned}$$

which taking $-\log$ results in:

$$l(a^n) = \log\binom{n}{n_1} + \log(n+1).$$

So we get the same result as in the previous example.

This mixture can be also implemented sequentially giving a plug-in approach to estimate θ independent from n on each step t as:

$$\frac{n_1!n_0!}{(n+1)!} = \prod_{t=0}^{n-1} q(a_{t+1}|a^t)$$

where

$$q(1|a^t) = \frac{n_1(a^t)+1}{t+2}.$$

Here $n_1(a^t)$ is the number of ones in the string. The formula for the conditional probability for zero is symmetrical. This is the Laplace rule of succession [17].

Moreover, when other weight functions are used to perform the mixture (for example Dirichlet's density) other plug in interpretations can be obtained like the Krichevsky-Trofimov estimator (see 1.9)

1.7 Models

Next we will discuss some of the most commonly used models for probability estimation in compression algorithms.

1.7.1 Memoryless models

Memoryless models (like the Bernoulli model mentioned in section 1.6) consider that all the symbols in the input string are statistically independent and thus it only needs to estimate a single probability distribution for all symbols. In this case a sequential implementation only needs to update this single probability distribution for each symbol read. A memoryless model for an alphabet of size α has $\alpha - 1$ free parameters.

This is the simplest model with minimal model cost. The drawback is that this model is not able to exploit all the statistical redundancies existing in most input strings and thus is not able to achieve good compression rates.

1.7.2 Markov models

Markov models are used to parametrize sources where the probability of the next symbol depends on a finite number of contiguous past observations. Markov models define a number of states which encode some history of the previous symbols. In each of these states there is a different probability distribution, so the Markov model has a set of conditional probabilities for the next symbol for each state.

The number of previous symbols on which the probability of the next symbol depends on is called the order of the Markov model. In general a Markov model of order k will have α^k states, one for each possible combination of the past k symbols. For each of these states a probability distribution has to be estimated when the model is used for encoding.

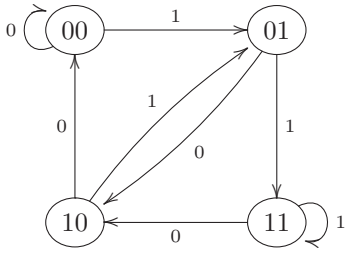
For example lets define a Markov model of order 2 for a binary alphabet with 4 (2^2) states $s_0 \dots s_3$ and a set of conditional probabilities such as:

- $s_0(0,0) \Rightarrow p(0|s_0) = 0.3$
- $s_1(1,0) \Rightarrow p(0|s_1) = 0.3$
- $s_2(0,1) \Rightarrow p(0|s_2) = 0.1$
- $s_3(1,1) \Rightarrow p(0|s_3) = 0.5$

where the conditional probability of the symbol 1 is not given but is the complement of the probability for the symbol 0.

To implement an adaptive coding scheme the model includes a transition function $f : S \times A \to S$ that, given a state in the state set S and a symbol emitted in this state, returns the state where the next symbol should be processed. Moreover, the statistics in the current state should be updated before the transition to account for the occurrence of the last symbol.

Under this definition it is possible to implement a Markov model with a *finite state machine* (FSM). For example the FSM for the Markov model from the previous example could be graphically represented like this:



In the case of a Markov model of order k we need to estimate $\alpha^k(\alpha - 1)$ parameters as there are $\alpha - 1$ conditional probabilities to estimate for each of the α^k states of the model.

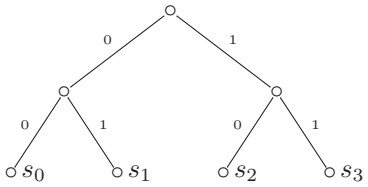
1.7.3 Tree models

Tree models are a parametrization of sources that may have finite memory $\leq k$ (e.g. Markov sources of order $\leq k$). A tree model for a source alphabet of size α is represented with a α-ary tree where every internal node has exactly α children (i.e. the tree is *full*). The edges leading to each of these

children are labeled with one different symbol of the alphabet. The node that is reached through an edge with label a is called the node *in the direction of* a. For every leaf in the tree there is also a set of conditional probability distributions on the source alphabet [37]. In principle this tree has depth k and α^k leaves, one for each state of the Markov source.

In many states the number of symbols needed to determine the next state may be less than the order of the model. Thus, merging nodes from the full balanced tree containing all possible contexts we get a model with a smaller number of free parameters (a smaller model class) and therefore a lower model cost [37].

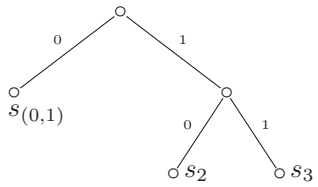
For example, the Markov source defined in the previous section would have this tree representation:



Remembering that the conditional probabilities in this model were:

- $s_0(0,0) \Rightarrow p(0|s_0) = 0.3$
- $s_1(1,0) \Rightarrow p(0|s_1) = 0.3$
- $s_2(0,1) \Rightarrow p(0|s_2) = 0.1$
- $s_3(1,1) \Rightarrow p(0|s_3) = 0.5$

it can be seen that in this source the states s_0 ans s_1 can be merged as they share the same conditional probabilities. In other words we can predict the next state with the same probability only knowing that the last symbol was 0 without the need to know the previous symbol. Taking this into consideration we can give another tree model for this source as:



The new state $s_{(0,1)}$ merges the two previous discrete states s_0 and s_1. Thus the importance of tree models lays in the use of less parameters than the corresponding Markov models. Usually the nodes of the tree models are called *contexts* and the model itself is called a *context tree*. The symbols in the path to each node are reversed relative to the order of the symbols in the input string to be able to shorten paths and reach a shorter context.

A common algorithmic tool to represent tree models is the construction of the so called *suffix trees*. A suffix tree for a string $u\$$ where $u \in A^*$ and the special symbol $\$ \notin A$ is an α-ary tree with the following properties:

1. The tree's edges are labeled with non-empty strings.
2. The concatenation of the labels in the paths from the root to each leaf have a one-to-one relationship with the non empty suffixes of $u\$$. Thus, there is a leaf for every non empty suffix of $u\$$ and all the tree's leaves are non empty suffixes of $u\$$.
3. All internal nodes have at least two children.

The \$ character is used to ensure that no suffix is prefix of another one. If this special character was not included property 2 could not be assured because each suffix which is a prefix of another would be a branching node and not a leaf.

This structure was first introduced in [38] and has been used for many different purposes like finding substrings in a string or pattern matching. It is in fact a radix tree (also called patricia tree)

[25, 14] for the suffixes of $u\$$. This same structure is called *Generalized context tree* in [22]. This structure is built using the reversed input string as the parameter so that the contexts represented by each leaf are reversed with respect to the order of the symbols in the input string.

The basic idea of the compression schemes that use context trees is to build a tree that contains information on the occurrences of each symbol in every context (possibly bounded by a maximum tree height). Here a *context* is defined as any suffix of a sequence a^t where the next symbol a_{t+1} occurs. A context of length n is the equivalent of a state on a Markov source of order n. Each node s of the tree is associated with the subsequence of source symbols that occurred after context s and this data is used to estimate the probability of the next symbol in this context.

Given this tree representation it is not clear how to efficiently find the next context to use after a symbol has been processed in one of the contexts of the tree. If a^i is the string processed so far, the simplest way would be to traverse the tree from the root following the edges with labels $a_i, a_{i-1}, a_{i-2}, \ldots$ until a leaf is reached. This algorithm is not practical because of the large number of tree traversing operations needed.

A solution for this has been proposed in [20] as the *FSM closure* of the tree model. Formally, a tree is said to have the *FSM property* if it defines a next state function f which given a state on the tree (i.e. a leaf) and a symbol from the alphabet returns a new state such that for any sequence a^{n+1} we have:

$$s(a^{n+1}) = f(s(a^n), a_{n+1})$$

where $s(a^n)$ is the state reached after processing the sequence a^n. The FSM closure of a context tree r is the smallest tree that has the FSM property and has r as a subtree.

Thus, the FSM closure endows the tree with next-state transition pointers that allow the encoder to jump to the node where the next symbol will be encoded in $O(1)$ time. As the FSM closure process adds nodes to the tree there are also *Origin* pointers that provide access to the original states of the tree where the statistics are stored and updated. An efficient algorithm to construct the FSM closure of a tree is also described in [20].

We will return to tree models later in section 1.8.

To summarize, the process of losslessly compressing a string needs a probability model that can assign probabilities to the string and a procedure to encode the data using these probabilities with a number of bits per symbol a as close to $-\log p(a)$ as possible.

1.8 Double universality

Based on Rissanen's lower bound, and given a class $\mathcal{C}$ of models we can define a code as *Universal with optimal convergence rate* if for each source in the class the normalized expected redundancy vanishes as $\frac{k \log n}{2n}$ up to lower order terms, where k is the number of free parameters and n is the input length. In this case the goal is to find universal codes with optimal convergence rate with respect to a union of a countable number of model classes of growing dimensionality: $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \ldots \cup \mathcal{C}_n \cup \ldots$ [24]. An example of this is the case where, for every k, $\mathcal{C}_k$ is the class of k-th order Markov sources over some alphabet.

A double universal code should have a code length close to the minimum over all the classes of the universal code for each individual model class. For example, for the class of Markov sources the code in [31] has a redundancy upper bounded by;

$$(\alpha - 1)\, \alpha^k \frac{\log n}{2n} + \frac{c(k)}{n} \quad \forall k$$

for an alphabet of size α, a source code of length n and a model of memory size k. Here $c(k)$ is independent of n.

As shown with universal codes, here we can also use both the plug-in and mixture approaches. In the plug-in approach we use a two part code whose first part is the code for an integer i which identifies the best model class. This can be identified by exhaustive search for example. The second part of the code would be a universal code for the model class described in the first part.

The mixture approach can be also achieved in two stages, first within each $\mathcal{C}_i$ and then over all the model classes.

Returning to the subject of tree models, it turns out that there exist efficient double-universal and sequential codes in the class of tree models of any size. For example there are various versions

of the *Context* algorithm [28] that use the plug-in approach and the CTW algorithm [39, 40] which produces a sequential probability assignment using a mixture of all the models in the class.

With the plug-in approach a particular context is selected for each symbol to be encoded and the occurrences of the symbols in this context are used to calculate the conditional probabilities. Using the mixture approach the probability assignment is a mixture of assignments of all possible contexts of different lengths.

There is also the semi-predictive approach [29, 20] where the best tree model is described to the decoder in the first pass and then the data is sequentially encoded using this tree in a second pass. In this case the tree is pruned before being sent to the decoder in order to use an optimal number of parameters.

1.9 Probability estimation

When a model is used to assign probabilities using formula (1.2) it has to be able to estimate $P(a|s)$ for every symbol a in each state s. This is accomplished by collecting the number of occurrences of each symbol in every state of the model and estimating the probability distributions from this data. These counts are updated as long as the symbols are processed both in the encoder and the decoder. In this section we discuss different approaches to estimating the probability distribution from the number of occurrences of each symbol.

For example, consider an alphabet with α symbols and let $c_s(a)$ define the number of occurrences of symbol a in a state s and let c_s be the sum of occurrences of all symbols in that state. One naïve approach would be to use an empirical frequency estimator assigning $\frac{c_s(a)}{c_s}$ as the conditional probability of symbol a. The problem with this idea is that we are assigning zero probability to the symbols that have not appeared in s and it is always possible that a new symbol occurs for the first time.

As an alternative to this the Laplace estimator [17] already mentioned in 1.6 can be used. In this case the probability of a known symbol would be $\frac{c_s(a)+1}{c_s+\alpha}$. In this setting the numerator would be one for the symbols that appear for the first time.

Another common estimator is the Krichevsky-Trofimov estimator [15, 16] (KT from now on), also mentioned in 1.6 which is similar to Laplace where the conditional probability of symbol a is $\frac{c_s(a)+1/2}{c_s+\alpha/2}$. This estimator has the additional property that there is a known bound for the parameter redundancy. For example for a Bernoulli source where $p(1) = \theta$ and $p(0) = 1 - \theta$ and a sequence a^n with n_0 zeros and n_1 ones it can be proved [39] that:

$$\log \frac{(1-\theta)^{n_0}\theta^{n_1}}{KT(n_0, n_1)} \leq \frac{1}{2}\log(n) + 1.$$

We will see how these estimators are used in the various compression algorithms described below.

Chapter 2

CTW

2.1 CTW algorithm

As mentioned in section 1.8 the CTW (Context Tree Weighting) [39, 40] algorithm works over the class of tree models using a mixture approach. The algorithm uses context trees with contexts up to a certain depth h.

On each context s the algorithm stores the number of bit occurrences in the subsequence associated with that node. This is done for contexts of every length in the interval $[0 \dots h]$

On the leaves of the tree the only available information are the symbol counts so it is assumed that the sequence associated to the leaves is memoryless and the KT estimator is used to assign a probability for that sequence. For an internal node s of the tree a recursive argument can be used assuming that there are good probability estimations for the subsequences associated with the two children of s (we are assuming a binary alphabet here). Namely, it is possible to use the product of the probability estimation on both children as the estimation for the parent node. Besides this, there is still the possibility of using the KT probability estimator with the counts stored in s. So the idea on internal nodes is to mix this two possibilities with $1/2$ coefficients.

It is sound to use this mixture (called *weighted distribution*) as it can be seen that if $P_1(a^t)$ is a good distribution estimation for source 1 and $P_2(a^t)$ is a good distribution for source 2 then:

$$P_w(a^t) \triangleq \frac{P_1(a^t) + P_2(a^t)}{2}$$

is a good probability distribution for both sources. As a proof for this it can be shown that, if an arithmetic coder is used, then:

$$\begin{aligned} l(a^t) &< \log \frac{1}{P_w(a^t)} + 2 \\ &= \log \frac{2}{P_1(a^t) + P_2(a^t)} + 2 \\ &\leq \log \frac{2}{P_i(a^t)} + 2 \\ &= \log(2) - \log\left(P_i(a^t)\right) + 2 \\ &= \log \frac{1}{P_i(a^t)} + 3 \end{aligned}$$

where $i \in \{1, 2\}$. So the codeword length only increases by one bit in the worst case.

In this way if the KT probability of the sequence with n_0 zeros and n_1 ones that occurs in node s is denoted as $P^s_{KT}(n_0, n_1)$ and the already calculated weighted estimation for the two children nodes (called $0s$ and $1s$) are named P^{0s}_w and P^{1s}_w respectively the CTW probability estimation P^s_w for this node would be:

$$P^s_w = \begin{cases} \frac{1}{2} P^s_{KT}(n_0, n_1) + \frac{1}{2} P^{0s}_w P^{1s}_w & \text{for } 0 < h(s) < h \\ P^s_{KT}(n_0, n_1) & \text{for } h(s) = h \end{cases}. \tag{2.1}$$

Here $h(s)$ is the height of the node and h is the maximum height of the tree, so when $h(s) = h$, s is a leaf.

Let $\mathcal{C}_h$ be the class of tree models of height not larger than h, and $K_h(r)$ be the cost of encoding a tree $r \in \mathcal{C}_h$ using a natural code (this code uses a bit per tree node, see page 41). Let also leaves(r) be the set of leaves of r. Considering this and given a node $s \in r$ it is also proved by induction in [39] that:

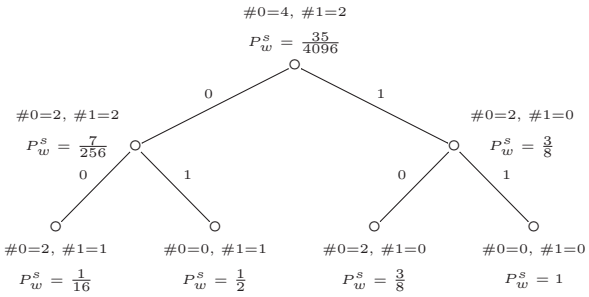
$$P_w^s = \sum_{r \in \mathcal{C}_{h-h(s)}} \left[2^{-K_{h-h(s)}(r)} \prod_{u \in \text{leaves}(r)} P_{KT}^{us}(n_0, n_1) \right]$$

with

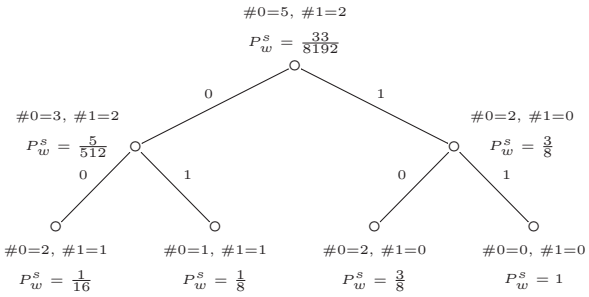
$$\sum_{r \in C_{h-h(s)}} 2^{-K_{h-h(s)}(r)} = 1.$$

This is to say that the weighted probability on a node of height $h(s)$ can be seen as a mixture of the estimated probabilities $P_{KT}^s(n_0, n_1)$ on the leaves of the sub-trees of height not larger than $h - h(s)$ weighted by the cost of such trees. Thus the value $P_w^\lambda(a_{h+1}^n | a_1^h)$ which is the weighted probability at the root of the tree is the probability assigned by CTW to the string a_{h+1}^n given a_1^h past symbols.

Let us show now an example for a context tree of depth 2. and an input string of 0010100. After processing the first 6 input symbols, and assuming an initial state given by the string 00, the tree nodes have the following values:



Then, after processing the last symbol (0 in context 01), the tree turns out to be:



The value P_w^λ on the root of the tree would be used to update the intervals in an arithmetic encoder. Here the whole sequence probability is used to update the coder intervals. Based on equation (1.3) we have that:

$$\begin{aligned} F(a^n) &= \sum_{x^n < a^n} P(x^n) \\ &= F(a^{n-1}) + \sum_{x < a_n} P(a^{n-1}x). \end{aligned}$$

Thus the root of the CTW tree gives us the value $P(a^{n-1}u)$ for each $u \in A$. In the case of a binary alphabet the value $P(a^{n-1}0)$ is calculated first as it is enough to update the arithmetic coder interval because:

$$F(a^n) = \begin{cases} F(a^{n-1}) & \text{if } a_n = 0 \\ F(a^{n-1}) + P(a^{n-1}0) & \text{if } a_n = 1 \end{cases}.$$

In case the next symbol is 1 the value $P(a^{n-1}1)$ is also calculated to keep the tree updated. The pseudo code for the CTW algorithm is given in algorithm 3.

Algorithm 3: CTW algorithm

input : String a^n to compress

for $a_i = a_1$ **to** a_n **do**
  Calculate $\tilde{P}^o_{KT} = P^o_{KT}(n_0 + 1, n_1)$ where o is the root of the tree
  for $a_j = a_i$ **to** a_{i-h} **do**
    Calculate $\tilde{P}^s_{KT} = P^s_{KT}(n_0 + 1, n_1)$ for $s = a_i, a_{i-1}, \ldots, a_j$
  end
  for $a_j = a_{i-h}$ **to** a_i **do**
    Calculate $\tilde{P}^s_w$ for $s = a_i, a_{i-1}, \ldots, a_j$ using $\tilde{P}^s_{KT}$ as in (2.1)
  end
  Update $\tilde{P}^o_w$ where o is the root of the tree
  Update arithmetic coder using $\tilde{P}^o_w$
  if $a_i = 0$ **then**
    Update symbol counts in the root of the tree
    for $a_j = a_i$ **to** a_{i-h} **do**
      Update context tree counts on node $a_i, a_{i-1}, \ldots, a_j$
    end
    for $a_j = a_{i-h}$ **to** a_i **do**
      Set $P^s_w = \tilde{P}^s_w$ for $s = a_i, a_{i-1}, \ldots, a_j$
    end
    Set $P^o_w = \tilde{P}^o_w$ where o is the root of the tree
  else
    Update symbol counts in the root of the tree
    for $a_j = a_i$ **to** a_{i-h} **do**
      Update context tree counts on node $s = a_i, a_{i-1}, \ldots, a_j$
      Calculate $P^s_{KT}(n_0, n_1 + 1)$
    end
    for $a_j = a_{i-h}$ **to** a_i **do**
      Update P^s_w for $s = a_i, a_{i-1}, \ldots, a_j$
    end
    Update P^o_w where o is the root of the tree
  end
end

2.2 Byte-oriented files compression

The CTW binary algorithm can be applied to byte-oriented files simply by treating bytes as sequences of bits. If we do this we can use only a few bits of a previous symbol as the context for a following one. This does not seem useful as in binary files the natural context of a symbol are the previous full 8 bit symbols.

The scheme is adapted for byte-oriented files in [43] using different trees for each bit of a byte. For each bit, a tree is selected from a group of trees using the previous bits of the byte as selectors. For example, for the fourth bit in a byte the three previous bits are used to select one tree from the eight (2^3) possible trees. The context tree for that bit would be formed by the symbols that preceded the previous bytes with the same three bit prefix as this one.

Also, 256-ary trees are used instead of binary trees and the contexts are formed by full bytes. Thus the CTW weighting is only performed on byte boundaries and not using bits inside of a symbol.

In this case the CTW weighting is:

$$P_w^s = \begin{cases} \frac{1}{2}P_{KT}^s(n_0, n_1) + \frac{1}{2}\prod_{0 \le i \le 255} P_w^{is} & \text{for } 0 < h(s) < h \\ P_{KT}^s(n_0, n_1) & \text{for } h(s) = h \end{cases}.$$

It is important to note that, even if the trees are not binary, the statistics stored in each node only count bit occurrences, it is only the contexts that are not binary. This has the advantage that the trees are smaller than full binary trees for the same context length and thus reduce the redundancy because the number of bits required for describing a 256-ary tree are less than the required for describing a binary tree (which would have more nodes).

2.3 Probability estimation

As was shown before, the CTW algorithm requires an estimation of the probability of the next symbol based on the counts of bit occurrences in each node. When doing text compression the KT estimator is not used but a variant called *zero-redundancy estimator* is used instead. If n_0 and n_1 are the counts of zeros and ones respectively this is defined as follows:

$$ZR(n_0, n_1) = \begin{cases} \frac{1}{2}P_{KT}(n_0, n_1) & \text{for } n_0 > 0, n_1 > 0 \\ \frac{1}{2}P_{KT}(n_0, 0) + \frac{1}{4} & \text{for } n_0 > 0, n_1 = 0 \\ \frac{1}{2}P_{KT}(0, n_1) + \frac{1}{4} & \text{for } n_0 = 0, n_1 > 0 \\ 1 & \text{for } n_0 = 0, n_1 = 0 \end{cases}$$

Intuitively this estimator takes half KT and divides the remaining probability space adding $1/4$ to the string with all zeros and $1/4$ to the string with all ones. This is so because many byte values never occur in ASCII text files so it is common that after particular bit combinations within a byte most of the symbols are ones or zeroes. As a result, for some of the context trees, many estimations end up working over states which contain only zeros or only ones.

More formally, with this estimator the redundancy that results for not knowing the parameter for a source that only generates zeros or ones is never larger than two bits whereas with the KT estimator this redundancy grows logarithmically with the number of symbols (see [43]).

2.4 CTW implementation

The implementation of CTW used for comparison is the one presented in [43] by Erik Franken and Marcel Peeters from Eindhoven University of Technology. This has several improvements over the original algorithm implementations. A general description of the implementation will be given below.

2.4.1 Tree construction

This implementation uses standard 256-ary suffix trees to store the statistics for each context and calculate the weighted statistics. In fact 255 of these trees are used as mentioned in section 2.2.

The trees are built incrementally adding nodes as new symbols are processed. The algorithm defines a maximum tree size parameter to keep the size of the trees bounded.

The trees are stored using a closed hash table in order to save the space needed for children pointers in a standard tree implementation. When the child in the direction of symbol a of node s needs to be found a hash function is evaluated. This hash function depends on the index of s in the table, the symbol a, the position of the current bit being processed in the ASCII byte (called *phase*) and the size n of the hash table as:

$$\begin{aligned} \mathit{offset} &= \text{hash}(a) \oplus (2(\mathit{phase} + 1)) \\ \mathit{index} &= \text{index}(s) + \mathit{offset} \mod n \end{aligned}$$

where *hash* is a random permutation table. The phase is needed because all 255 trees are stored in the same table and this decreases the probability that two different trees conflict with each other,

If the node in this index does not coincide with the searched node it is said that a *collision* has occurred. In this case a new index is computed, and then another if there are still collisions. The next index is computed by adding offset again to the last result. If the node cannot be found after a certain number of tries the node cannot be stored in the hash and the algorithm continues with a smaller tree. This means the value of h is reduced for some contexts.

One problem with this strategy is that the hash table size must be defined before starting the compression process. This is a hard task because the optimal size is not easily predicted and memory will be wasted if the table is too large and it may not be possible to build the full context tree if the table is too small.

2.4.2 Weighting arithmetic

A few arithmetic techniques are used to achieve better performance in this implementation.

The first one is the use of integer arithmetic for all the operations. Integer arithmetic is much faster than floating point and the results do not depend on the floating point implementation of the architecture. In order to diminish the magnitude of rounding errors all values are multiplied by a constant.

In addition, all values are represented with their logarithmic values. In this way all the multiplying and dividing operations become adding and subtracting. However adding the logarithmic values is harder; for this the jacobian logarithm is used:

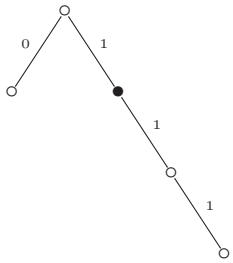
$$\log(p_1 + p_2) = \log(p_1) + \log(1 + \frac{p_2}{p_1}) = \log(p_1) + \log(1 + 2^{\log(p_2) - \log(p_1)})$$

To increase the efficiency of the logarithm operation the algorithm uses precalculated tables for $\log(x)$ and the jacobian logarithm $\log(1 + 2^x)$.

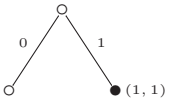
Finally the algorithm stores some quotients of probabilities in each node which allows this implementation to calculate the weighted probabilities using only the nodes on the current path instead of using additional nodes of the tree (see [43]). This quotients are updated every time the context appears in the input string.

2.4.3 Path pruning

Another technique used to improve the performance and space requirements of the algorithm consists in pruning from the tree the paths that, after certain level, only have one child for every node. Those are called unique paths and the probability in every node is the same as the estimated probability on the first node of the path. Unique path pruning decreases the number of nodes on the trees saving memory and reducing the number of nodes to search and the number of calculations needed. This same technique is also used in other algorithms which use context trees. As an example, in the following tree all nodes below the black node would be pruned.

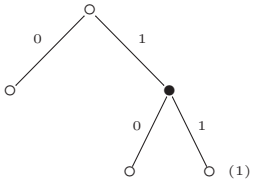


When a new context is seen that coincides in part with a pruned path, the coinciding part of the original path has to be reconstructed using the data from input string that must be saved in memory. This reconstruction process requires saving the already processed symbols in a memory buffer and also a reference to this input string for every pruned path in every pruned node. After the previous figure tree is pruned and the pruned path reference is stored on the node it would result in:



The authors found that the memory saved by pruning tree nodes is greater than the space lost to the input string buffer. When very large files are processed and the buffer gets full it is frozen and pruned contexts are not reconstructed when they occur after the buffer is full.

In the previous example a new child of the black node in the direction of 0 would leave the tree as shown in the next picture after reconstructing one pruned node from the input buffer:



Chapter 3

PPM

3.1 PPM algorithm

The basic idea of PPM (*Prediction by Partial Matching*) [8] is again to use the last few symbols in the input string (the context of the symbol) to predict the next one. PPM uses Markov models of order k where each state is the context of length k where each symbol is found. In fact several models of different order are used and one of them is chosen every time a symbol is coded. If a higher order model cannot make a prediction then a lower order model is used (this is called a *partial match*). This is the same idea as the plug-in approach for universal codes except that the context selection is performed by heuristic methods. So, this algorithm lacks the universal properties that Context (chapter 5) and CTW (chapter 2) possess.

As shown in section 1.7.3 a tree model can be used instead of the set of Markov models of different orders to lower the model cost. In this case, the leaves of the tree are not the only possible states as internal nodes are also considered as states for lower order models. Thus all the Markov models of order $1 \dots k$ are represented in a single tree structure. Furthermore it is only necessary to look at the parent node of a context to find the next shorter context for a symbol.

In every state of each Markov model a count of the number of occurrences of each symbol in that context is stored. These counts are used to estimate the symbol probabilities which, in turn, are used with an arithmetic coder. The probability estimation can be made with any of the methods mentioned in section 1.9 or others. If the symbol did not occur in a state both the encoder and the decoder have to backtrack to a lower order model. This is signaled using a special symbol called *escape*. To ensure that a symbol never seen before can be encoded properly a special last model of order -1 must be added. This last model assumes that all alphabet symbols have equal probability and therefore there are no escapes.

This, in fact, can be also considered as a way of creating a mixture of several models assigning a different weight to each one of them. In this case, supposing that the escape probability of the context of length k is e_k and h is the length of the longest context that can make a prediction, the weight of each context is:

$$w_k = (1 - e_k) \prod_{i=k+1}^{h} e_i \qquad -1 \leq k < h$$
$$w_h = (1 - e_h)$$

where the weight of the model of length k is the probability of descending to that model and not going any further. So, the final contribution of a model of order k to the mixture probability can be calculated as the product of the escape probabilities of every model of order greater than k multiplied by the complement of the escape probability in the model of order k. These weights will define a convex combination assuming all escape probabilities are between 0 and 1 and the escape probability in the state of order -1 is 0.

To show that the weights define a convex combination we need to show that they sum up to 1. We can prove this by induction on h. In the case that $h = 0$:

$$w_0 + w_{-1} = (1 - e_0) + e_0 = 1$$

Assuming the property is true for $h = d - 1$ we have:

$$(1 - e_{d-1}) + \sum_{t=0}^{d-2}(1 - e_t) \prod_{i=t+1}^{d-1} e_i + \prod_{j=0}^{d-1} e_j = 1$$

Then when $h = d$:

$$(1 - e_d) + \sum_{t=0}^{d-1}(1 - e_t) \prod_{i=t+1}^{d} e_i + \prod_{j=0}^{d} e_j =$$

$$(1 - e_d) + (1 - e_{d-1})e_d + e_d \left(\sum_{t=0}^{d-2}(1 - e_t) \prod_{i=t+1}^{d-1} e_i \right) + \prod_{j=0}^{d} e_j =$$

$$(1 - e_d) + (1 - e_{d-1})e_d + e_d \left(e_{d-1} - \prod_{j=0}^{d-1} e_j \right) + \prod_{j=0}^{d} e_j = \quad \text{(induction hypothesis)}$$

$$1 - e_d + e_d - e_{d-1}e_d + e_d e_{d-1} - \prod_{j=0}^{d} e_j + \prod_{j=0}^{d} e_j = 1.$$

It makes sense to use the complement of the escape probability as the weight of a model as the escape probability is the probability that a particular context will be followed by a symbol that has never seen before. Contexts with high escape probability should have less weight as their predictions are not very good.

To formally define the probability assigned to a symbol using this algorithm let $c_k(a)$ be the number of times symbol a has occurred in a context of order k. We assume that $c_{-1}(a) = 1 \; \forall a \in A$. Let us also define c_k as $\sum_{a \in A} c_k(a)$.

The probability that will be assigned to the *escape* symbol remains to be defined. This is usually determined heuristically and there are several different policies to do this [8, 2]:

PPMA With this method the probability of symbol a in a model of order k is:

$$p_k(a) = \frac{c_k(a)}{c_k + 1}$$

and the escape probability for a novel symbol in a model of order k is:

$$e_k = 1 - \sum_{a \in A_k} p_k(a) = \frac{1}{c_k + 1}$$

where

$$A_k = \{a : c_k(a) > 0\}$$

PPMB In this approach a symbol is not predicted in a context until it has been seen at least twice. The probability of symbol a is:

$$p_k(a) = \frac{c_k(a) - 1}{c_k}$$

Then the escape probability is:

$$e_k = 1 - \sum_{a \in A_k} p_k(a) = \frac{|A_k|}{c_k}$$

where

$$A_k = \{a : c_k(a) > 1\}$$

PPMC This is similar to PPMB but makes predictions as soon as the symbol is seen in the context and the probability of symbol a results in:

$$p_k(a) = \frac{c_k(a)}{c_k + |A_k|}$$

The escape probability is:

$$e_k = 1 - \sum_{a \in A_k} p_k(a) = \frac{|A_k|}{c_k + |A_k|}$$

and A_k is defined just like PPMA.

Most PPM schemes use a technique called *exclusion* [2] in order to simplify the calculations needed to obtain the probability of a symbol. Using *exclusion* only the model with the largest context is used to predict every new symbol. However, if the new symbol has not appeared before in this context the *escape* code is transmitted to let the decoder know that the longest context will not be used and the algorithm falls back to the model with next smaller context of the symbol. When a context is found that can predict the symbol only the probability of that context is used ignoring (i.e. excluding) all the lower order models. In this case, instead of doing a mixture of the different models in the class, the *exclusion* technique selects a single model for each symbol as in the plug-in approach.

A problem with the implementation of this algorithm is that every time a symbol is seen in a context the counts for the symbol must be updated in all the shorter prefix contexts which can be very time consuming if the model is large. To cope with this problem when exclusions are used, there exists another technique called *update exclusion*. In this way when a symbol's probability can be estimated in one context the counts of the symbol in the shorter contexts are not updated (i.e. are excluded). This can be rationalized supposing that the correct statistic to collect for the lower order models is not the raw count but rather the frequency with which a symbol occurs when it is not predicted by a longer context. This is mainly done as a way to simplify the algorithm implementation but in fact it can improve the compression rate (for example in [2] they mention improvements of about 2%).

To implement update exclusions efficiently we need a way to find the context where the next symbol will be coded without the need to traverse the tree from the root to find it. This usually means that we need an additional function $NextState(s, a_i)$ that given node s and symbol a_i returns the node where the probability of symbol a_{i+1} will be estimated. This is the longest path on the tree for the context of symbol a_{i+1} and is the first node where the symbol will be processed. If escapes occur the final node where the symbol is coded will be different.

Usually the context tree is constructed incrementally as the input symbols are processed sequentially. The same procedure that builds the tree creates the pointers needed to implement the $NextState$ function.

The pseudo-code for the PPM algorithm with update exclusion assuming that the root of the tree is the state where all symbols have equal probability is shown in algorithm 4.

Algorithm 4: PPM algorithm with update exclusion

input : Symbol a_i
input : Current node of the tree: s
output: The probability for a_i
output: Node to use to encode the next symbol

```
repeat
    if symbol a_i is predicted in s then
        Update symbol counts on s
        return The probability for a_i predicted in s
        return NextState(s, a_i)
    else
        Update symbol counts on s
        s = Parent(s)
    end
until s = root
return The probability for a_i predicted in root
return NextState(s, a_i)
```

3.1.1 SEE

Many algorithms based on PPM use a technique called *Secondary Escape Estimation* (SEE for short) [6] that tries to improve estimations of the escape probability on states where few symbols have been processed. In these states usually the escape probability is overestimated initially as the escapes are very common in the first iterations of the algorithm.

The idea is not to use the number of escapes in these nodes *per se* to estimate the escape probability. Instead some properties of the nodes are used to find a SEE context where the escape probabilities of all nodes with the same properties are accumulated. As the SEE context counts data from different nodes it is expected that it can give a more precise escape estimation than each of the nodes by themselves.

The properties of the node used to determine the SEE context are determined heuristically considering parameters like:

- The number of escapes in the node
- The number of processed symbols in the node
- The height of the node
- The number of symbols processed in the parent node

These values are quantized to a fixed number of bits and joined to form the number of the SEE context to use. The SEE context gives an escape probability and then the counts of this context are updated (along with the statistics in the original tree node) considering if an escape was generated or not. In general the SEE context statistics are updated using large increments so initial data is rapidly forgotten by new updates.

For example in [6] the SEE context for a node is determined by:

- The depth of the node mapped to two bits as:

$$\begin{cases} \text{depth}/2 & \text{if depth} \leq 6 \\ 3 & \text{otherwise} \end{cases}$$

- The number of escapes in the node using two bits (if greater than 3, SEE is not used)
- The total number of symbols found in the state (denoted here by c_s) mapped to three bits as:

$$\begin{cases} c_s & \text{if } c_s \leq 2 \\ 3 & \text{if } c_s \in \{3,4\} \\ 4 & \text{if } c_s \in \{5,6\} \\ 5 & \text{if } c_s \in \{7,8,9\} \\ 6 & \text{if } c_s \in \{10,11,12\} \\ 7 & \text{otherwise} \end{cases}$$

- The 6^{th} and 7^{th} bits of the last four ASCII symbols in the input string which act as a flag to identify lower case (11), upper case (10), punctuation (01) and escape (00) characters.

3.2 PPM implementations

There are a few relatively modern algorithms based on PPM which claim to have improvements over the original algorithm. Some of them are:

- PPMZ by Charles Bloom [6]
- The algorithm presented in [9] by Michelle Effros
- PPMd by Dmitry Shkarin [35]

Among these the best one in terms both of speed and compression is PPMd. This was the algorithm studied and used for practical comparison in this work.

3.2.1 PPMd

PPMd uses the same basic probability assignment discussed in the case of PPM with *exclusion* with some heuristic modifications regarding the counts of symbols. It also uses a different tree structure and a form of SEE that will be discussed below. The tree is built incrementally as input symbols are processed.

Tree construction

Assuming that we want to estimate the probability of symbol a using the counts in context s, the procedure to select the final context and update the tree as needed is shown in algorithm 5. We let $c_s(a)$ be the number of occurrences of symbol a in node s, suf_s is the context which represents the suffix of s one symbol shorter and $f_s(a)$ is the symbol that follows a in the input string after the first occurrence of a in s after s was created. The value $f_s(a)$ is obtained by associating to node s the index in the input string of the first occurrence of a in the node. The symbol in the next index is $f_s(a)$. $f_s(a)$ is null if a has not occurred in s. The tree is initialized with a single root node.

In short the tree is built following this procedure:

- A new child s' is created when a new symbol a occurs for the second time in a context s (line 6 of algorithm 5). $c_{s'}(a) = 0 \; \forall a \in A$.
- This procedure is repeated for all suffixes of s until the root of the tree is reached (line 40) or a suffix context is found where a new node addition is not necessary because it already has a child in the direction of a (line 21) or no new node is added (line 31).
- All these newly created nodes need to have their suffix pointer correctly initialized. The suffix of each of these nodes is the new node created on the suffix of its parent node (line 9) or the child of the suffix of the parent in the direction of a in case it already exists (line 23) or the suffix of the parent node in case a child is not created (line 33) or its parent (line 7) in case it is not changed later. This only happens if the parent is the root.
- If we let s_{new} be a new node created, after that we found its suffix, (denoted here s') the counts for the symbol $f_{s'}(a)$ in s_{new} are initialized to the value b which is a heuristic value based on the number of symbols and the sum of their frequencies in s' (lines 10, 25, 27, 34).

As shown in algorithm 5 the PPMd tree is not constructed as a traditional context tree as was shown before. Here the different context nodes are added to the tree in the same order that the symbols appear in the input string. In this case a path from the root to a node represents a substring from the input that occurred more than once. The context of a symbol occurring in a node s can be found traversing from s to the root (ie. they are backwards with respect to the traditional context tree). Thus, the context to estimate the probability of a new symbol is found traversing the tree instead of using an external link to provide this function.

On the other hand, this construction does not allow the algorithm to reach a shorter context traversing the tree towards the root. In order to achieve this the algorithm keeps suffix links which point to the node of the tree where the suffix of each node is stored. Suffix links are used as auxiliary structures in many algorithms, like the ones used to build suffix trees from [23, 36] and also in the algorithm to build the FSM closure of a tree in [20].

This structure requires less pointers than the traditional PPM tree construction with "short-cut" links to the next state. The former structure only needs one pointer in addition to the standard children pointers while the latter needs an additional array of pointers for the next-state link. On the other hand this tree's edges cannot be compressed because all edges have length one.

The algorithm has a limit on the amount of memory that the tree model can use. By default the model is grown until this level is reached and then it is restarted from scratch. There are also options to freeze the model and to try to cut it. This last option is considered by the authors to be too slow.

We next show an example building a tree for the string 011011. We label each node with its name and the pair (n_0, n_1) with the number of symbols 0 and 1 that occurred in each node. The dashed arrows represent the suffix pointers. We also assume for simplicity that the symbol counts in new nodes are always set to 1 ($b = 1$).

The first two symbols are processed in the root node (s_0) and no new nodes are created as the symbols are distinct.

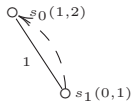
Algorithm 5: PPMd tree construction algorithm

input : Symbol a
input : Selected context s (a node of the tree)

ok = **false**
s_{new} = **null** `/* This will be a new node created. A reference is needed as we search for its suffix context */`
repeat
  $c_s(a) = c_s(a) + 1$
  if $c_s(a) = 2$ **then**
    Create new node s' as a child of s in the direction of a
    $\text{suf}_{s'} = root$ `// this may change later`
    if $s_{new} \neq$ **null then**
      $\text{suf}_{s_{\text{new}}} = s'$
      $c_{s_{\text{new}}}(f_{\text{suf}_{s'}}(a)) = b$ `/* b is based on the number of symbols and the sum of their frequencies on` $\text{suf}_{s'}$ `(see information inheritance below) */`
    end
    for $i \in A$ **do**
      $c_{s'}(i) = 0$
    end
    $s_{\text{new}} = s'$
    if $s = root$ **then**
      $c_{s'}(f_s(a)) = b$
    else
      $s = \text{suf}_s$
    end
  else if $\exists\ s' =$ *child of* s *in the direction of* a **then**
    if $s_{new} \neq$ **null then**
      $\text{suf}_{s_{\text{new}}} = s'$
      if $f_{s'}(a) \neq$ **null then**
        $c_{s_{\text{new}}}(f_{s'}(a)) = b$
      else
        $c_{s_{\text{new}}}(f_{\text{suf}_{s'}}(a)) = b$
      end
    end
    ok = **true**
  else
    if $s_{new} \neq$ **null then**
      $\text{suf}_{s_{\text{new}}} = s$
      $c_{s_{\text{new}}}(f_s(a)) = b$
      ok = **true**
    else
      $s = \text{suf}_s$
    end
  end
until $s = root$ **or** *ok*

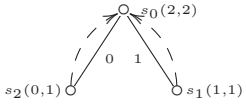
011011

The third symbol gets $c_{s_0}(1) = 2$ so a new node is created as s_1 (line 6 of algorithm 5). As $\text{suf}_{s_1} = s_0$ (line 7) and $f_{s_0}(1) = 1$ then $c_{s_1}(1) = 1$ (line 17). The selected context to estimate the next symbol will be s_1.



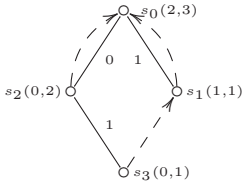
011011

The fourth symbol is a 0 and we have $c_{s_1}(0) = 0$ so it is searched in the suffix s_0 (line 37) and, as $c_{s_0}(0) = 2$ a new node (s_2) is created (line 6). $c_{s_2}(1) = 1$ because $\text{suf}_{s_2} = s_0$ (line 7) and $f_{s_0}(0) = 1$.



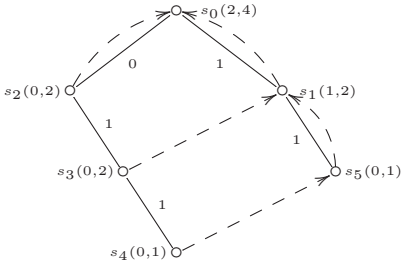
011011

The next symbol is a 1 and, as now $c_{s_2}(1) = 2$, a new node is created (line 6) as s_3. The suffix of the parent of s_3 (s_0) already has a node in the direction of 1 (s_1) so the suffix pointer is initialized pointing to this node (line 23). The value $c_{s_3}(1)$ is set to 1. As this is the first symbol 1 that occurs in s_1 ($c_{s_1}(1)$ was initialized to 1 when s_1 was created) the algorithm uses $f_{s_0}(1)$ (line 27). This is the third symbol of the string and it is 1.



011011

The last symbol is again a 1 so $c_{s_3}(1) = 2$ and a new node is created in the direction of 1 as s_4 (line 6) and also a new node (s_5) is created in the suffix of the parent (s_1) to create the proper suffix links. The count of occurrences of the symbol is incremented in all nodes in the path to the root.



011011

Coding

When a symbol a occurs in a state s where $c_s(a) > 0$ its probability is estimated using the statistics stored on s. If $c_s(a) = 1$, a new node is created as was explained earlier. In case $c_s(a) > 1$, s should have a child in the direction of a and that child node will be selected to estimate the probability of the next symbol.

If $c_s(a) = 0$ an escape is emitted and the algorithm tries to encode the symbol in suf_s. This search continues emitting escapes in each state until the root is reached or a state s' is found where $c_{s'}(a) > 0$. After s' is found all the other states visited during the search are updated to include the new symbol a and more nodes are added as needed. The selected context for the following symbol is the child in the direction of a of s' (or the root if there is no s')

Symbol and escape probabilities

In order to calculate symbol and escape probabilities the algorithm keeps a tally of the number of times a symbol a occurs in a state s ($c_s(a)$) and another counter for the total number of symbol and escape occurrences in s (denoted $SummFreq_s$).

If we, as before, let $A_s = \{a : c_s(a) > 0\}$ in the first state tried when coding a symbol the symbol and escape probabilities are given by:

$$p_s(a) = \frac{c_s(a)}{SummFreq_s}$$

$$e_s = 1 - \frac{\sum_{a \in A_s} c_s(a)}{SummFreq_s}.$$

In other states $SummFreq_s$ is not used and this value is substituted by a value obtained from SEE (denoted by SEE) resulting in:

$$p_s(a) = \frac{c_s(a)}{SEE}$$

$$e_s = 1 - \frac{\sum_{a \in A_s} c_s(a)}{SEE}.$$

Information inheritance

Another difference in the PPMd algorithm is the use of what the authors call *Information Inheritance*. The underlying idea is that the statistics stored in a node must depend in some form on the statistics of the suffix. This is why, when a new node s is created, some information is obtained from suf_s in order to initialize the counts of occurrences of symbols in s with values different from zero. This is the b value in lines 10, 25 and 34 of algorithm 5. Given a new node s and its suffix suf_s the value b for symbol a is computed as:

$$a_0 = SummFreq_{\text{suf}_s} - |A_{\text{suf}_s}| - c_{\text{suf}_s}(a) - 1$$

$$b = \begin{cases} 1 & \text{if } a_0 > 5(c_{\text{suf}_s}(a) - 1) \\ 2 & \text{if } 2(c_{\text{suf}_s}(a) - 1) \leq a_0 \leq 5(c_{\text{suf}_s}(a) - 1) \\ \dfrac{(c_{\text{suf}_s}(a) - 1) + 2a_0 - 3}{a_0} & \text{otherwise} \end{cases}$$

The value a_0 is roughly the number of symbols different from a that occur in suf_s. If this number is much larger than $c_{\text{suf}_s}(a)$ then the algorithm assumes that there are several symbols in suf_s and b is smaller. Otherwise the algorithm assumes that there are not many symbols different from a in suf_s and then uses a larger number for b as it can be estimated to be more probable in s as it has followed the symbol a before.

For this same reason when a symbol a is coded in a state s the value $c_{\text{suf}_s}(a)$ is also incremented. The value i of this increment is given by:

$$i = \begin{cases} 2 & \text{if } |A_{\text{suf}_s}| > 1 \text{ and } c_{\text{suf}_s}(a) < 115 \\ 1 & \text{if } |A_{\text{suf}_s}| = 1 \text{ and } c_{\text{suf}_s}(a) < 32 \\ 0 & \text{otherwise} \end{cases}$$

This increment can change the estimation of symbols that are coded in suf_s and also in future children of suf_s that can be created. The idea is to put a limit on the number of occurrences of the symbol and not increase it if the symbol has already occurred many times. The limit is lower if the symbol s is the only one that occurred in suf_s as in this case its probability will already be very high and increasing the number probably does not help very much.

Chapter 4

Other compression algorithms

4.1 BWT

4.1.1 BWT algorithm

There is a class of algorithms, which are based on the BWT (Burrows-Wheeler Transform) [7], which do not process the whole input string sequentially but divide the input in blocks and work over each of them separately. The idea is to apply the BWT to each of these blocks and compress each of them individually. This transformation tends to group equal characters together so the resulting string is easier to compress using simple compression algorithms. This transformation is reversible so the original data can be restored by the decoder.

The BWT is composed by two quite different algorithms: the direct transformation that converts the input string in the transformed string and the reverse transformation which obtains the original string from the transformed one. The direct transformation is shown in algorithm 6.

Algorithm 6: BWT direct transformation

input : String a^n
output: String u^n and integer l

1 M = new $n \times n$ matrix
2 **for** $i = a_1$ **to** a_n **do**
3 $\quad M[i]$ = the cyclic permutation of a^n that starts with a_i
4 **end**
5 `Sort(`*the rows of matrix* M`)`
6 u^n = the last column of M
7 l = the index in M so that $M[l] = a^n$
8 **return** (u^n, l)

For example, applying this algorithm to the string *cocos* will result in the matrix:

$$\begin{pmatrix} c & o & c & o & s \\ c & o & s & c & o \\ o & c & o & s & c \\ o & s & c & o & c \\ s & c & o & c & o \end{pmatrix}$$

then the BWT will be the pair $(socco, 1)$

Algorithm 7 shows the reverse transformation.

Continuing with the previous example for the string *cocos* the input string u^n would be *socco* and the value of l is 1. Then the array z is $(5, 3, 1, 2, 4)$ and we get:

Algorithm 7: BWT reverse transformation

input : String u^n and integer l
output: String a^n

```
1  v^n = Sort(u^n) // v^n is the first column of M
2  for i = 1 to n do
3  |  if u_i is the j^th occurrence of symbol w in u^n then
4  |  |  z[i] = k where v_k is the j^th occurrence of symbol w in v^n
5  |  end
6  end
7  for i = 0 to n − 1 do
8  |  a_{n−i} = u_{z^i[l]}
   |  // z^0[x] = x
   |  // z^{i+1}[x] = z[z^i[x]]
9  end
10 return (a^n)
```

$$a_5 = u_1 = s \ (z^0[1] = 1)$$
$$a_4 = u_5 = o \ (z^1[1] = 5)$$
$$a_3 = u_4 = c \ (z^2[1] = 4)$$
$$a_2 = u_2 = o \ (z^3[1] = 2)$$
$$a_1 = u_3 = c \ (z^4[1] = 3)$$

so we recover the original string $x^n = cocos$.

One example of a simple compression algorithm that is used after applying the BWT is *move-to-front* coding [4] which keeps a list of the symbols in the string being processed moving each symbol that appears in the input to the front of the list. Each symbol is coded by its index in this list so symbols that occur more often tend to use smaller indexes and thus require less bits to encode. In particular if there are large clusters of the same symbol in the input string (as will be the case when the BWT is used) all of them will be coded by the number 0.

Also it is customary to apply a *run-length* coding algorithm after the move-to-front coding. Run-length coding works by replacing sequences of repeated symbols by one of those symbols followed by the number of repetitions.

Summarizing, the BWT of a string v of n characters proceeds by building all the cyclic permutations of v and sorting them lexicographically. The result of the transformation is the last character of all the n ordered permutations (string u) and the position l of the original string in the ordered list of cyclic permutations. The position l is needed by the decoder in order to reverse the transformation and obtain the original string. The reordered sequence is then encoded in a second pass without resorting to use an arithmetic coder. See [10] for a complete analysis of the algorithm.

To understand why the BWT algorithm works we can see that if a character is commonly followed by the same symbols, when the permutations are sorted there will be a region on u where the character will occur disproportionately with respect to the rest of the string. For example, think of the letter t in the word *the*. All the cyclic permutations that start with *he* will be lumped together resulting in a region of u with many occurrences of the symbol t along with other symbols that normally precede *he* like s, T and S.

Thus, the BWT of a reversed string groups symbols that follow the same context, so the algorithm can be considered as coding based on a context tree. The difference is that there is no context selection procedure and only a reordering of the sequence grouping symbols in similar contexts. In this view, if the input string is not reversed the context of a symbol can be seen as formed by the symbols that follow it instead of the symbols that precede it. This does not work in an algorithm that attempts to process the input sequentially but works in BWT as it processes the input in blocks.

4.1.2 Efficient implementation of BWT

In the case of the direct transformation procedure clearly the most complex step is the sorting of the cyclic permutations. This could be done in $O(n^2 \log(n))$ time (assuming the comparison of strings takes $O(n)$ time) using an algorithm like quicksort, but this problem can be reduced to that of sorting all the suffixes of a string. This could be efficiently accomplished by building a suffix tree of the input string and doing an inorder traversal of this tree. This can be done in linear time using an algorithm like the ones described in [23, 36] to build the suffix tree.

The reverse transformation procedure only sorts a string instead of sorting n strings so it is inherently faster than the direct one. It can be implemented in three passes over the input string and a pass over an auxiliary string of the size of the alphabet (see [7] for a full description).

It is important to note that neither of the two algorithms require the use of floating point arithmetic in contrast with all the other algorithms mentioned in this thesis. This and the relative simplicity of the algorithms make BWT compression inherently faster that other algorithms.

4.1.3 BWT implementation

The BWT implementation studied here is *bzip* by Julian Seward [32, 33]. It does not use a suffix tree in order to sort the block permutations but builds a suffix array. In this context, a suffix array is simply an array which indexes all suffixes in lexicographic order.

This suffix array is built sorting the suffixes using the Bentley-Sedgewick ternary quicksort [5] and using some optimizations which include pre-sorting the first characters of each prefix using a form of counting sort (see appendix A) similar to the one used in the algorithm described in page 38 and other further optimizations described in [32].

In order to compress the string after the BWT is applied *bzip* uses move-to-front and then run-length coding as was explained in section 4.1.1. After these steps the algorithm uses Huffman coding [21] to assign shorter codes to more probable symbols (see [32] for details).

4.2 Context mixing algorithms

A context mixing algorithm combines the predictions of several different models using weighted averages [18]. This seems similar to the ideas behind CTW (or the mixture approach of universal codes) but is more general in the sense that it mixes predictions from arbitrary models instead of only using context trees. This of course lacks the universality properties proved for CTW as the model class is much larger. Besides, this algorithm mixes the probabilities predicted for each symbol instead of doing a combination of several probabilities for the entire string as is done in CTW.

In this case the weights are adjusted dynamically in order to favor the most accurate models for the data being coded and the compression ratio achieved becomes better as the number of different models used is larger. The trade-off is that this also augments the processing time and hardware resources needed. Top ranked context mixing compressors can be 500 to 1000 times slower than efficient practical compressors like *gzip* [18]. Also, if many models are used, there would be a cost in code length during the time the algorithm adapts itself to ignore models that make incorrect predictions and gives more weight to the more accurate models.

Context mixing works as follows in a binary setting: for each bit to encode each model m independently outputs two numbers $n_{0m}, n_{1m} >= 0$ which can be seen as a measure of the probability estimated by the model that the next symbol will be 0 or 1 respectively. These numbers in general do not need to sum up to one so the probabilities are in fact $\frac{n_{0m}}{n_m}$ and $\frac{n_{1m}}{n_m}$ where $n_m = n_{0m} + n_{1m}$. Then, a weighted summation of n_{0m} and n_{1m} over all the models is performed in order to obtain the final algorithm prediction. These same steps are done during the decoding stage so that the same probabilities are recovered and an arithmetic coder can be used. In this way letting w_m be the weight of model m we get:

$$
\begin{aligned}
S_0 &= \epsilon + \sum_m w_m n_{0m} \\
S_1 &= \epsilon + \sum_m w_m n_{1m} \\
S &= S_0 + S_1 \\
p_0 &= \frac{S_0}{S} \\
p_1 &= \frac{S_1}{S}
\end{aligned}
$$

where p_0 and p_1 are the probabilities that the next symbol will be a 0 or a 1 respectively and ϵ is a small constant used to guarantee that S_0 and S_1 are greater than zero.

The values of w_m can be initialized to favor previously known better models or with the same value for every model. After each symbol is processed, assuming its value was x (0 or 1), the weights are updated as follows:

$$
w_m = \max\left(0, w_m + \frac{(S\, n_{1m}) - (S_1\, n_m)}{S_0 S_1}(x - p_1)\right). \tag{4.1}
$$

In (4.1) the term $|(x - p_1)|$ can be seen as the error measure for the current prediction. It will be 0 if the prediction is correct and go up to 1 in case of a completely wrong prediction ($p(x) = 0$). The term $(x - p_1)$ will be positive when $x = 1$ and negative otherwise. We can also see that:

$$
\begin{gathered}
(S\, n_{1m}) - (S_1\, n_m) = \\
((S_0 + S_1)n_{1m} - (S_1(n_{0m} + n_{1m})) = \\
S_0 n_{1m} - S_1 n_{0m}.
\end{gathered}
$$

So this term will tend to be positive for models that predict 1 with more probability than their peers and negative in the opposite case. This coincides with the sign of $(x - p_1)$ to raise the weight of models that make correct predictions.

Current context weighting implementations use dozens of models. Some of them are context based, others use word models and some use bidimensional models to model bidimensional data like images or databases. Some implementations also use a form of SEE (called SSE in this case as we are estimating probabilities of symbols not escapes) after mixing the estimations while others also use a SSE stage internal to each model in addition to the one used after the mixing stage.

The implementation of this algorithm used for comparison is zpaq 1.02 by Matt Mahoney [44]

Chapter 5

Context

5.1 Context algorithm

The Context algorithm, introduced by Rissanen in [28], is a lossless universal coding scheme in the class of tree models which uses the plug-in approach. The algorithm builds a tree that stores the number of symbols occurring in every context and then a distinguished context from the tree is selected for each symbol to code. The statistics on this node are used for estimating probabilities. The algorithm does not collect symbol counts for every possible context as this would require too much space. Only some of the contexts are used and, for choosing those, the algorithm associates a cost with each context and only includes a particular context if its share in reducing the conditional entropy exceeds its cost.

This algorithm was created as an improvement over the Lempel-Ziv algorithm [42] considering that LZ partitions the string in incremental segments and can not capture dependencies between non contiguous symbols which arise, for example, in image compression applications. To address this problem the context algorithm collects statistics on overlapping sets of symbols each one of them defining a *context* on which the symbol occurrences can be conditioned.

It has been proved [37] that this coding scheme is universal in the class of tree sources with the following upper bound for the normalized difference between the ideal code length assigned by any finite memory source with k sets of equivalent states and the ideal code length assigned by this algorithm (for a string of length n):

$$\frac{k(\alpha-1)}{2n}\log n + O\left(\frac{1}{n}\right)$$

which goes to zero at the fastest possible rate (considering Rissanen's bound) as n goes to infinity.

This is how the tree is built in [37] (the algorithm used in [28] is slightly different because it always constructs full trees):

1. The algorithm starts with only the root node with its symbol counts all zeroed.

2. Recursively let $r(t)$ be the last constructed tree and a_{t+1} the next observed symbol. The next tree $r(t+1)$ is generated by climbing the tree $r(t)$ following the path determined by $a_t, a_{t-1}, a_{t-2}, \ldots$ taking the branch left for 0 and right for 1. For each node visited the count for symbol a_{t+1} is incremented by one until the deepest node of the tree is reached. Lets call this node $a_t, a_{t-1}, \ldots, a_{t-j+1}$.

3. If the last updated symbol count becomes at least 2, a new node in the direction of a_{t-j} has to be created. Let us call this node $a_t, \ldots, a_{t-j}$. This new node has all its symbol counts with value zero except for symbol a_{t+1} whose count is set to one. This resulting tree would be $r(t+1)$.

The goal of this stage is to build all the relevant contexts (the tree will grow only in directions where repeated symbols occur) and accumulate symbol statistics for each one of them in a practical way. In this case we are assuming that the context of a symbol is comprised by the past contiguous

symbols, but in general the algorithm can use any computable function of the past symbols as this context.

So far we have described the tree construction but the procedure used to select a distinguished context for each input symbol remains to be defined. There are several variants of this procedure but the basic idea is to select the node which would have assigned the shortest code length for its symbol occurrences in the past string. One intuitive choice is to set the cost of a node to be a function of the code length for the symbols encoded in that node and then estimate the probability of symbol a_{t+1} using the counts on the longest context for a_{t+1} on $r'(t)$ where $r'(t)$ is the tree with minimum cost after observing symbol a_t.

In [20], the model cost is calculated as if the full context tree for the entire string would be constructed and then in a second pass the cost of the nodes would be determined and the tree would be pruned. As in this algorithm the tree must be sent to the decoder the cost of each node also includes the number of bits required to send each additional node to the decoder.

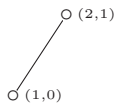
If we let $c_s(a)$ be the number of occurrences of symbol a in node s, the complete pseudo-code for the Context algorithm is shown in algorithm 8.

Algorithm 8: Context algorithm

input : Symbol a_i and context $a_{i-1}, \ldots, a_{i-h}$
output: The probability for a_i

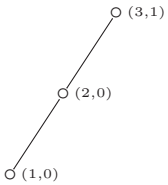
s = root of $r(i-1)$
$j = 1$
while *s has a child in the direction of a_{i-j}* **do**
  $c_s(a_i) = c_s(a_i) + 1$
  s = child of s in the direction of a_{i-j}
  $j = j + 1$
end
$c_s(a_i) = c_s(a_i) + 1$
if $c_s(a_i) \geq 2$ **then**
  Create new node s' as a child of s in the direction of a_{i-j}
  $c_{s'}(a) = 0 \quad \forall a \in A$
  $c_{s'}(a_i) = 1$
  $r(i) = r(i-1) \cup s'$
else
  $s' = s$
  $r(i) = r(i-1)$
end
s'' = `SelectContext` $(a_{i-1}, \ldots, a_{i-j}, s')$ `/* Explained in the text above */`
return probability of symbol a_i estimated on node s''

As an illustration of the tree construction process we consider the tree corresponding to the binary string 10001101. After the first two symbols are processed (10) no new nodes have been constructed and the root node of $r(2)$ has count $(n_0, n_1) = (1, 1)$. Then, when the next 0 is processed the count n_0 for the root node becomes 2 so a new node is added to the left of the root node with count $(1, 0)$ so $r(3)$ is:



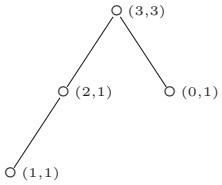
10001101

The same thing happens with the next 0 (context starting with 00) as the count of the node added in the previous step reaches $(2, 0)$ leaving the following tree $r(4)$:



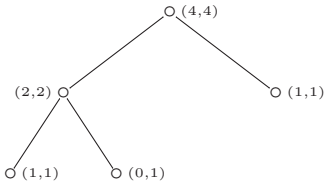
10001101

The next 1 symbol (context starting with 00) only alters the counts of the nodes but the leaf had $n_1 = 0$ so no new nodes are added. The second 1 does make a difference because its context now starts with 1 so a new node is added to the right of the root leaving the following tree as $r(6)$:



10001101

Now the next 0 (context starting with 1) just alters the counts on the root and the last added node but the last 1 symbol whose context starts with 01 generates the following tree as $r(8)$:



10001101

5.2 Semi-predictive context algorithm

In this work, a new context algorithm using the semi-predictive approach has been developed. It is based on the algorithm defined in [20] which has the important practical property of having linear encoding and decoding time.

Generally speaking the encoding process in the algorithm in [20] has these steps (pseudo-code shown in algorithm 9):

1. Build a suffix tree of the complete reversed input string.
2. Prune this tree in order to obtain an optimal model (a tree that minimizes the code length).
3. Build the FSM closure of the pruned tree. The construction of the FSM closure of the tree is done for computational reasons as was explained in section 1.7.3.
4. Use this tree as a model to obtain probabilities and use them with an arithmetic encoder.

The decoder receives the optimal tree used in the encoding procedure with added nodes in order to form an atomic[1] full tree. It is done in this way because a full tree can be encoded using less bits than would be required to encode an arbitrary tree. Additional bits could be used to describe the exact tree used by the encoder (simply by stating for every node if it belongs to the tree used by the encoder or not) but this cost can be avoided while preserving a linear complexity algorithm.

[1] A tree is atomic if all the labels on its edges have length one

Algorithm 9: Context semi-predictive encoding algorithm

input : String to encode: a^n
output: The pruned tree
output: The probability for a_i, $1 \leq i \leq n$

```
r = BuildSuffixTree (a^n) // builds the suffix tree for a^n
foreach node s in a DFS traversal of r do
    if HasChildren(s) then
        if Cost(s) < Σ_{i ∈ Child(s)} Cost(i) then
            Make s a leaf
        end
    end
end
return pruned tree r
r' = BuildFSM (r) // builds the FSM closure of r
s = root of r'
for a_i = a_1 to a_n do
    return probability of symbol a_i estimated on node s
    c_s(a_i) = c_s(a_i) + 1
    s = NextState(s, a_i) /* Follows the pointer to the next state given by the FSM
        closure */

end
```

Considering this, the decoding process in [20] consists of the following steps (pseudo-code shown in algorithm 10):

1. Build a new tree starting from the one sent by the encoder deleting all leaves and nodes whose outgoing degree after deleting the leaves is one.
2. Build the FSM closure of this new tree (this tree has fewer nodes than the tree used in the encoder because of the nodes deleted in step 1).
3. Decode the input string using an arithmetic decoder with this tree as a model adding the missing nodes as necessary.

Algorithm 10: Context semi-predictive decoding algorithm

input : The pruned tree r
input : The output of the arithmetic encoder for the input string a^n
output: Symbol a_i, $1 \leq i \leq n$

```
r' = DeleteLeaves (r)
r'' = BuildFSM (r') // builds the FSM closure of r'
s = root of r''
for i = 1 to n do
    return symbol a_i decoded using the probabilities on node s
    c_s(a_i) = c_s(a_i) + 1
    s = NextState(s, a_i) /* Follows the pointer to the next state given by the FSM
        closure */

end
```

5.2.1 Important contributions

Next we will describe the algorithm developed in this thesis. The main innovations are:

- Regarding the implementation:
  - This algorithm uses a modified version of the *wotd* algorithm (write-only top-down) presented in [11] in order to build the suffix tree and prune it at the same time (so we implement steps 1 and 2 of the encoding algorithm in a single step). The top-down approach of *wotd* guarantees that once a branch has been created, it will never be traversed again by the algorithm thus allowing us to prune the branch during the suffix tree construction. This way the complete suffix tree is never built allowing memory savings and thus enabling the algorithm to process longer input files using the same amount of memory. Also in practice the procedure takes less time.
  - The algorithm defines a constant indicating the maximum height of the pruned suffix tree. In this way all nodes higher than this constant are automatically pruned without calculating the cost which is a quite complex function (see below). If this constant is chosen sufficiently large this does not affect the compression ratio and diminishes the work needed. If the constant is too short the tree will not be optimal and the compression rate will be lower.
  - It has the first implementation of the ideas presented in [19] to implement the decompression using an incomplete FSM closure. Some extra steps are added in the implementation because of the use of the PPM techniques mentioned next.
- Regarding the compression:
  - The algorithm has been adapted to use some techniques seen in PPM algorithms such as update exclusions and SEE.
  - The prunning procedure was changed to account for the use of the mentioned PPM ideas.
  - The KT estimator was replaced for another more adequate for large alphabets.

Next we will describe the encoding and decoding procedures with more detail.

5.2.2 Encoder

Initialization

The first task during the encoding procedure is to output the alphabet of the input string in order to send it to the decoder. The encoder and decoder do not necessarily need to know the alphabet (they can assume the alphabet is composed by all 256 8-bit symbols) but the probability estimations are better if they know that some symbols never occur in the string. With a smaller alphabet all symbols can have higher probabilities and thus lead to a shorter compressed representation.

Suffix tree building

The next step of the encoding procedure is to build a suffix tree (see 1.7.3) for the reversed input string ($\overline{u}$). We build the tree for $\overline{u}$ because the suffixes of this string are the prefixes of u. These prefixes are the contexts where we need to store the number of occurrences of each symbol to estimate probabilities.

Although there are linear time algorithms to build suffix trees (see [23, 36]) our algorithm uses, as was mentioned before, a modified version of the *wotd* algorithm in order to build the suffix tree and prune it at the same time. The linear algorithms presented in [23, 36], require more bytes per node to implement and also have very little locality of memory reference which is a problem in cached processor architectures.

The *wotd* algorithm uses an auxiliary array called *suffixes* that contains all suffixes of $\overline{u}\$$ ordered by increasing suffix length so:

$$\begin{aligned} \mathit{suffixes}[0] &= \overline{u}\$ \\ &\dots \\ \mathit{suffixes}[n] &= \$. \end{aligned}$$

The algorithm begins the construction of the suffix tree by creating nodes for all the children of the root and then each of these nodes is processed and new nodes are created for its children. These new nodes created are processed recursively in a DFS fashion.

The children of the root node are found sorting the *suffixes* array by the first symbol of each suffix using a counting sort algorithm (see appendix A). A group of suffixes that start with symbol a (which are lumped together by the sorting process) corresponds to an edge whose label starts with a outgoing from the root. If there is only one prefix starting with a the edge leads to a leaf and otherwise it leads to a branching node. The label on edges that lead to a leaf is the rest of the string up to the symbol \$. This is not an atomic representation of the tree per definition of a suffix tree (given in section 1.7.3).

Note that this step of the procedure is remarkably similar to the one used in the BWT implementation presented in [32]. In that case it is used during the sorting phase of the BWT algorithm (see section 4.1.3) which is very different to this one but it shows that both algorithms are handling contexts in a similar way.

To recursively process a branching node whose label starts with a (leaves do not need further processing) the algorithm first computes the length of the edge label. This is found by calculating the least common prefix (*lcp*) of all the suffixes in the group. This is the longest prefix that all suffixes in the group share.

When this has been found the edge is built and all suffixes in the group are shortened by the length of the *lcp* discarding the prefix symbols that all suffixes share and this group is processed in the same way as was explained for the root.

The pseudo-code of this algorithm is shown in algorithm 11. Here we assume that the edge's labels are given by a pair of references to the complete input string $(begin, end)$. We also assume that the counting sort algorithm returns a structure we can use to get the indexes in the sorted *suffixes* array where the first and last occurrence of each symbol appear. Let us call this structure *csort*. For a symbol a we can get the beginning of the interval as $csort[a].begin$ and the end as $csort[a].end$. If the symbol a does not occur in the array to sort then $csort[a].begin = csort[a].end = -1$. The first invocation of this recursive algorithm is performed with the following input parameters: $((1, |\overline{u}\$|), root)$ where *root* is the root node of the tree to build.

Algorithm 11: *wotd* algorithm

input : The bounds of the interval of *suffixes* to process $(begin, end)$
input : Root node of the tree r
output: Suffix tree r for the suffixes in the specified region

```
1  csort = Sort (suffixes, begin, end)
2  foreach symbol a with csort[a].begin >= 0 do
3  |  if csort[a].begin < csort[a].end then
4  |  |  lcp = Lcp (suffixes[csort[a].begin], suffixes[csort[a].end])
5  |  |  subNode = New branching node from r with label
   |  |  (suffixes[csort[a].begin], suffixes[csort[a].begin] + lcp)
6  |  |  for i = suffixes[csort[a].begin] to suffixes[csort[a].end] do
   |  |  |  // Remove first |lcp| symbols
7  |  |  |  suffixes[i] = Substring(suffixes[i], |lcp|)
8  |  |  end
9  |  |  Wotd (csort[a].begin, csort[a].end, subNode)
10 |  else
11 |  |  New leaf from r with edge label starting with suffixes[csort[a].begin]
12 |  end
13 end
14 return r
```

The *wotd* algorithm uses a highly optimized tree structure to represent the suffix tree that is built. This structure and more details on the algorithm implementation are discussed in page 49.

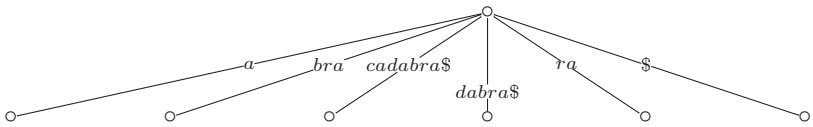
Lets us now show an example building the suffix tree for the string *abradacabra*. As shown we need to build a suffix tree for *abracadabra*\$.

The first step is to sort all suffixes of *abracadabra*\$ by its first symbol:

abracadabra\$
acadabra\$
adabra\$
abra\$
a\$
---
bracadabra\$
bra\$
---
cadabra\$
---
dabra\$
---
racadabra\$
ra\$

\$

We added horizontal lines to separate the suffixes that start with each one of the six symbols in the string. Thus we know that the root of the suffix tree has six children and three of them are leaves as there is only one suffix starting with c, d and \$. The *lcp* for the other tree branching nodes is "a", "bra" and "ra" respectively so the tree we can build so far is:



We continue recursively with the first branching node (the one labeled "a"). Here we sort again the suffixes in this interval discarding the first symbol "a" as this is the *lcp*. We then get:

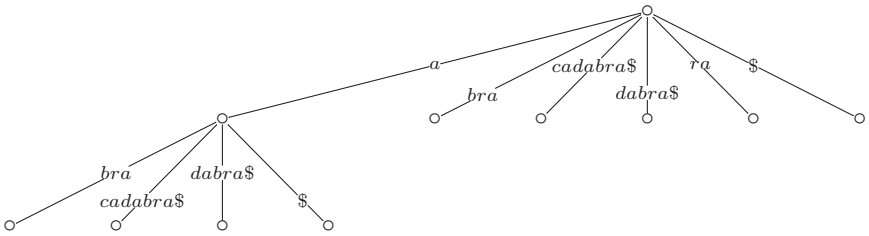
(a)bracadabra\$
(a)bra\$

(a)cadabra\$

(a)dabra\$

(a)\$

We have four children with three leaves and another branching node. For this branching node the *lcp* is "bra". So the tree we currently can build is:

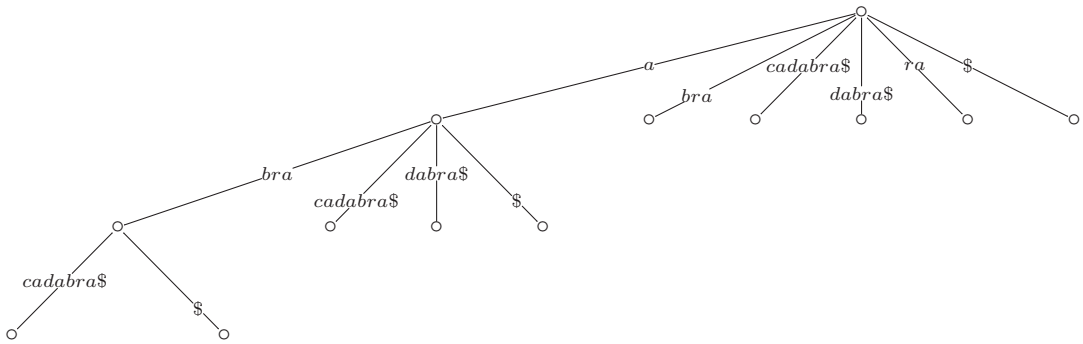


Following the next branching node the sorted prefixes are:

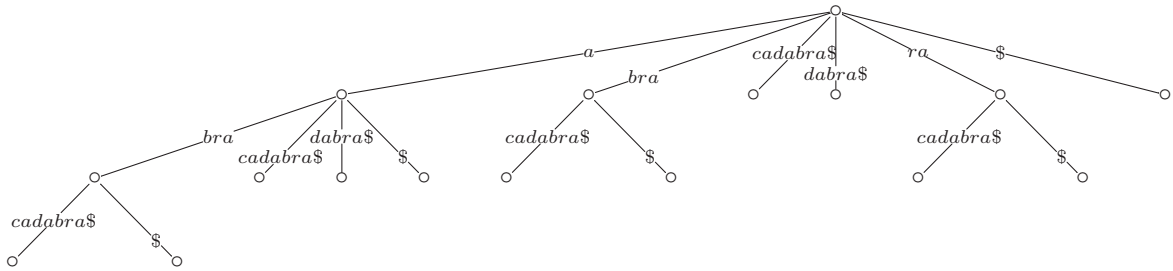
(abra)cadabra\$

(abra)\$

Both are leaves so the tree now is:



We then backtrack to process the two remaining branching nodes on the first level. As both contain only leaves we will next show the complete context tree:



Pruning

The pruning procedure seeks to build a tree r that minimizes the code length for a string a^n given by:

$$L(r, a^n) = L_r(a^n) + K(r)$$

where $K(r)$ is the cost of encoding the tree r in order to send it to the decoder and $L_r(a^n)$ is the code length of the input string conditioned on the states of r. It is important to remember that, as mentioned in section 5.2, the tree is converted into a full atomic tree when sent to the decoder, so $K(r)$ is calculated considering this augmented tree.

The value $K(r)$ depends on the algorithm used to encode the tree. In [20] the tree is encoded using a natural code which uses one bit per node where every bit indicates if the node is internal or if it is a leaf. In our algorithm the tree is encoded using the following procedure:

- The encoder sends the number r_i of internal nodes of the tree to the decoder. The decoder then can calculate the total number of nodes which is $r_i\alpha + 1$
- If we let r_l be the number of leaves in r, an arithmetic coder is used to encode the class of strings with r_i ones and r_l zeroes. The tree is traversed in-order and each node is encoded with the arithmetic encoder using the probability given by the class (ie. the probability of being an internal node or a leaf) and then r_i and r_l are adjusted to account for the symbol just encoded. Initially we have that $r_l = r_i(\alpha - 1) + 1$. As these values are adjusted the probabilities are given by:

$$P(\textit{internal}) = \frac{r_i}{r_i + r_l}$$
$$P(\textit{leaf}) = \frac{r_l}{r_i + r_l}$$

- This continues until there is no more nodes to encode or it can be determined that all remaining nodes are leaves.

Considering the previously mentioned probabilities of a node being internal or a leaf, the entropy of this probability distribution is:

$$H\left(\frac{r_i}{r_i\alpha+1}\right) = H\left(\frac{1}{\alpha+\frac{1}{r_i}}\right) \sim H\left(\frac{1}{\alpha}\right)$$

where H is the binary entropy function. It follows that, using an arithmetic encoder, each node of the tree would need approximately $H\left(1/\alpha\right)$ bits.

On the other hand, the most important component of $L(r, a^n)$ is the code length for the symbols encoded in the tree's nodes. Let this value be $\kappa(a^n, s)$ for node s. The formula to calculate this value depends on the probability estimation used by the algorithm implementation and will be discussed below.

The algorithm which implements the pruning of the tree r is based on the observation that, by recursively assigning costs to sub-trees, an optimal tree consists of optimal subtrees. Specifically, for every node s, the sum of the costs of the optimal subtrees rooted at the children of s is compared with the cost of making s' a leaf where s' is a prefix of s and $|s'| = |Parent(s)| + 1$ (or $s' = \lambda$ if $s = \lambda$). This means that all the edges leading to leaves are made atomic as it makes no sense to consider a longer prefix as this will increase the cost of encoding the tree with no gains in compression. This is the same idea behind unique path pruning in the implementation of CTW described in section 2.4.3.

It is also necessary to remember that the tree that will be sent to the decoder will be atomic and full. Thus, there will be more nodes added to the tree and the cost of these new nodes also has to be taken into consideration.

Using these ideas in [20] every sub-tree rooted at a node s of r is assigned the cost $\Omega(s)$ recursively defined by:

$$\begin{aligned}
&\Omega(s) = min(\\
&\quad \alpha(|s| - |s'|)H\left(\frac{1}{\alpha}\right) + (\alpha - deg(s))H\left(\frac{1}{\alpha}\right) + H\left(\frac{1}{\alpha}\right) + \sum_{w \in CHILD_r(s)} \Omega(w), \\
&\quad \kappa(a^n, s') + H\left(\frac{1}{\alpha}\right) \\
&\quad)
\end{aligned}$$

where the first term represents the cost of leaving this node with all its children (and also all the added nodes due to completion) and the second term is the cost of pruning the tree transforming s' into a leaf. The first two summands of the first term count the number of nodes that appear when the tree is converted to a full atomic tree, the third is the cost of this node and the fourth is the sum of the optimal costs of the sub-trees rooted at each of the node's children. The term $H\left(\frac{1}{\alpha}\right)$ represents the cost of encoding a node of the full atomic tree as was shown before.

The previous expression can be simplified to:

$$\begin{aligned}
&\Omega(s) = min(\\
&\quad \alpha(|s| - |s'| + 1)H\left(\frac{1}{\alpha}\right) + H\left(\frac{1}{\alpha}\right) + \sum_{w \in CHILD_r(s)} \left[\Omega(w) - H\left(\frac{1}{\alpha}\right)\right], \\
&\quad \kappa(a^n, s') + H\left(\frac{1}{\alpha}\right) \\
&\quad)
\end{aligned}$$

and then defining a new variable $\Omega'(s) = \Omega(s) - H\left(\frac{1}{\alpha}\right)$ and subtracting the term $H\left(\frac{1}{\alpha}\right)$ we get:

$$\Omega'(s) = min\left(\alpha(|s| - |s'| + 1)H\left(\frac{1}{\alpha}\right) + \sum_{w \in CHILD_r(s)} \Omega'(w),\ \kappa(a^n, s')\right)$$

We then have that, if s is the root of r, $\Omega'(s) = L(r, a^n)$. This cost formula was also modified slightly in our algorithm on account of the specific details of the encoding procedure used. This will be explained in detail shortly.

In order to implement this pruning procedure some statistics must be gathered for every node. These statistics consist of the number of times each symbol is seen on that node. These counts are calculated recursively as the sum of the counts on the children of the node. The recursion starts at the leaves where the counts are initialized during the suffix tree construction as there only occurs one symbol which is the one that precedes that context in the reversed input string. The estimated code length of the leaves can also be readily computed as $\log(\alpha)$ as all symbols have equal probability in a leaf. Considering there is only one symbol the cost is then 0. Here and in the rest of the description of the algorithm we will consider A to be the subset of the alphabet that actually occurs in the input string and α to be its size.

These statistics are accumulated over the whole tree using this algorithm:

- For the leaves, statistics are composed by only one symbol with one occurrence whose context is represented by this node but they are not stored in our implementation
- For branching nodes the statistics are built recursively adding the symbols from the statistics of all the children of the node. The statistics for the leaves are only computed when they are needed to calculate $\Omega'(s)$.

As was mentioned before the algorithm used to build the suffix tree allows to prune the tree at the same time it is being built. This is achieved by evaluating $\Omega'(s)$ for every node just after calculating the statistics for the node. As the *wotd* algorithm guarantees that a branch will not be modified once it has been constructed the node can be readily pruned at this stage if necessary. Also, $\Omega'(s)$ is not evaluated for nodes after certain height as these nodes are always pruned.

FSM closure

The next step in the process is to build the FSM closure of the pruned tree obtained in the last section. The algorithm was already discussed in section 1.7.3.

Encoding

The encoding stage starts once the optimal tree and its FSM closure have been obtained using the previous steps. The procedure starts at the root of the tree where the first symbol of the input string is processed. Each symbol is encoded using the statistics stored in the current node of the optimal tree (via the *Origin* pointers explained in section 1.7.3). Then these statistics are updated to account for the new encoded symbol and the current node is changed following the FSM transition for the last processed symbol. This gives a new node where the procedure starts again until the input string is finished.

This is the basic encoding procedure but our scheme has a few differences in order to improve compression rates. The first one applies in the cases where the zero frequency problem appears. This is when a new symbol a has to be encoded in a state where it has never been seen before. In this case the algorithm encodes an escape symbol and does a PPM-like upwards search (see section 3) in the tree looking for a shorter context that predicts a. The encoder produces escape symbols until the root of the tree is reached or a state is found where a has been seen before. If a state with a positive number of occurrences of a is found a is assigned the probability given by this state. If the root is reached first and a is not found there an escape is emitted and then an order -1 state is used where all symbols are assigned the same probability.

During this search the symbols not seen in previous states are not considered because, when an escape is emitted in one state, the decoder knows that none of the symbols that occur in that state is the one we are looking for. These symbols are no longer considered (i.e. are *masked*) for the rest of the tree traversal. For each upwards traversal a set of masked symbols is recorded by the algorithm adding to it all the symbols present in each state visited. The symbols in this set are considered to have count 0 in further states.

This algorithm also uses the same *update exclusion* technique from PPM so that statistics are only updated on the nodes traversed by the search and are not altered in lower order contexts once the symbol is found. See algorithm 12.

The next issue is the determination of the escape probability in each state. The escape is considered as an extra symbol in each state and its probability is estimated differently in each one of them. As before we let $c_k(a)$ be the number of times symbol a has occurred in a context c of order k. Let

us also define A_k as the set of characters predicted in a node of order k minus the symbols masked for having already been seen in higher order contexts and c_k be $\sum_{a \in A_k} c_k(a)$. It is important to note that A_k is not constant for a node because it depends on the symbols already masked in the current search.

The symbol probability is defined as:

$$p_k(a) = \frac{2c_k(a) - 1}{2c_k}$$

and the escape probability for a novel symbol in a node of order k is:

$$e_k = 1 - \sum_{a \in A_k} p_k(a) = \frac{|A_k|}{2c_k}.$$

We also use a state of order -1 as in PPM to code symbols that were not found in any of the contexts present on the tree (ie. a symbol that occurs for the first time). In this case the symbol probability is:

$$p_{-1}(a) = \frac{1}{|A_{-1}|} \; \forall a \in A_{-1}.$$

This is a variant of the PPMC escape technique from section 3.1 called PPMD in [12]. In this case the sum of occurences of symbols and escapes is incremented by one for each symbol which makes it easier to estimate the code length for the pruning step. This is because the above probabilities can be implemented by increasing the count by $\frac{1}{2}$ for both the symbol and the escape in the first occurrence of the symbol (an escape is generated in this case) and by increasing the count of the symbol by 1 in subsequent occurrences (no escape).

The encoding procedure includes another feature which is the capability to reset the counts of occurrences of symbols in each state when their sum exceeds some predefined threshold. This is useful because it helps to maintain the statistics updated in case the input string is non-stationary (has different statistics in different sections). If the string statistics are constant along the whole string resetting is a bad idea because it forces the encoder to relearn the same statistics again. On the other hand if the string statistics are not uniform the resetting procedure helps the encoder to forget the old statistics in order to learn the new ones. In practice resetting statistics is more useful in data files than in text files so the algorithm uses different resetting thresholds in each case (the number of different symbols in the input file is used as a heuristic for distinguishing the two kinds of files).

Considering this we define the formula used to estimate the code length on a node s in this algorithm. Here $c_s(a)$, c_s and A_s are the symbols and counts of them for the whole input sequence for node s. We have

$$\kappa(a^n, s) = \log \frac{(c_s - 1)!}{\prod_{a \in A_s} \left[\Gamma\left(c_s(a) - \frac{1}{2}\right)\right] \pi^{-\frac{|A_s|}{2}} \left(\frac{1}{2}\right)^{|A_s|-1} (|A_s| - 1)!}.$$

More insight regarding the rationale behind this formula is given in appendix B.

The fact that symbols that occur for the first time in a context generate an escape and are passed to the parent of the current node also changes the pruning procedure. This is caused by the fact that the cost of not pruning a node is not only the sum of the costs of the children but also has to include the cost of the symbols that will have to be encoded at the parent when escapes are produced on its children. To account for this difference a term T is added to the previous minimization formula:

$$\Omega'(s) = min\left(\alpha(|s| - |s'| + 1)H\left(\frac{1}{\alpha}\right) + T + \sum_{w \in CHILD_r(s)} \Omega'(w),\ \kappa(a^n, s')\right)$$

where

$$T = \log \frac{(c'_s - 1)!}{\prod_{a \in A'_s} \left[\Gamma\left(c'_s(a) - \frac{1}{2}\right)\right] \pi^{-\frac{|A'_s|}{2}} \left(\frac{1}{2}\right)^{|A'_s|-1} (|A'_s| - 1)!}$$

and

$$c'_s(a) = \sum_{w \in CHILD_r(s)} 1_{A_w}(a),$$
$$A'_s = \bigcup_{w \in CHILD_r(s)} A_w,$$

where 1_{A_w} is the indicator function defined in 1.3. Thus T is calculated with the same formula as $\kappa(a^n, s)$ except that the number of occurrences of a symbol in this case is the number of child nodes where the symbol occurs as there will be one escape per symbol per child. In the same fashion the number of different symbols will be the number of different symbols in all children. Since the first (and only the first) occurrence of a symbol in a state will generate an escape that will be processed in the parent node, this is the exact cost generated by the use of escape symbols.

When the algorithm decides to reset the statistics on some node (the exact reset threshold will be discussed in page 52) it halves the counts of every symbol:

$$c_s(a) = \left\lfloor \frac{c_s(a)}{2} \right\rfloor \ \forall a \in A_s.$$

This raises another issue because some symbols which previously had a positive count are left with zero occurrences. This does not seem to be important but, as will be seen below, generates some problems during the decoding procedure. To solve these issues, when this situation of vanishing symbols arises, the ancestors of the rescaled node are processed and the counts of symbols that disappeared in the original node are also set to zero in case the parents do not have symbols that occurred more than once. These nodes only processed escapes and thus their probabilities were not used to encode any symbols. We will return to this problem when the decoding procedure is described.

Another optimization added to this basic algorithm is the use of a form of SEE in a similar way to the one used in the PPMd algorithm (see section 3.2). The idea behind this is to use the SEE context when the statistics gathered so far in the current state are not enough to obtain a good probability estimation. The SEE context groups statistics from all contexts with similar properties thus giving a better estimation.

In this case 2^{14} contexts are used and each context stores two values, denoted (SEE_1, SEE_2). When SEE is used we have that the probability of an escape is given by

$$e = \frac{SEE_1}{SEE_2}$$

and the probability of a symbol is given by:

$$p(a) = \frac{p_s(a)(SEE_2 - SEE_1)}{SEE_2}$$

where $p_s(a)$ is the probability assigned to the symbol by the current context counts ignoring the count of escapes (they are not used to assign escape probability in this case). In order to choose a context also chosen by other similar contexts a few properties of the current context are used in the selection:

- Three bits for the number of escapes
- Three bits for the number of occurrences of non-masked symbols and escapes in the state
- Three bits for the number of masked symbols in the state
- Two bits for the number of symbols in the parent node (only for text files)
- Three bits for the previous encoded symbol

These values where selected so that contexts with similar characteristics would share the same SEE contexts. Some heuristic tests were performed to select them.

To reduce a number to a fixed number of bytes we take the logarithm of the number. If the logarithm size exceeds the number of bits we use the maximum number available.

Neither the reset of statistics nor the use of SEE are reflected in the pruning formula as adding these variables make the formula very complex and not practical to implement.

If we let A_s be the subset of A composed by the symbols that occur in s the pseudocode for encoding one symbol is given in algorithm 12.

Algorithm 12: Context encoding

input : Symbol to encode: a
input : State s in tree r
output: The probability for a_i

```
1  masked = ∅ // empty set
2  ok = false
3  repeat
4      s = Origin(s)
5      if a ∈ A_s or Order(s) = −1 then
6          if UseSEE() then
7              see = FindSeeState()
8              return p(a) estimated on SEE state see
9          else
10             return p(a) estimated on node s
11         end
12         c_s(a) = c_s(a) + 1
13         masked = ∅
14         ok = true
15     else
16         if UseSEE() then
17             see = FindSeeState()
18             Code escape using counts on see
19         else
20             Code escape using counts on s
21         end
22         c_s(a) = c_s(a) + 1/2
23         c_s(escape) = c_s(escape) + 1/2
24         masked = masked ∪ A_s
25         s = Origin(Parent(s))
26     end
27 until ok
```

5.2.3 Decoder

Initialization

Here the only work to do is to decode the alphabet. This is done with the arithmetic decoder with the inverse procedure as that already explained in page 38.

Tree building

The tree building procedure is very simple in this case. The tree is constructed by the reverse process from the one showed in page 41. We obtain from the arithmetic decoder the information of whether the current node is a leaf or an internal node and build the tree recursively.

As the tree is being built we ignore the leaves and delete nodes whose outgoing degree after deleting the leaves is one. The leaves are never created during the recursive tree construction process and, after building each child of a node, the algorithm checks if the child has outgoing degree one and, if this is the case, this child node is deleted creating an edge with a label of length two.

Remember from section 5.2 that the tree that was encoded is the optimal tree obtained by the encoder (denoted r) with additional nodes added to form an atomic full tree, denoted r_{full}. r_{full} has more nodes than r, so its FSM closure will also have more nodes than the optimal tree closure (r'). When the leaves and nodes with outgoing degree one are deleted from r_{full} we end up with a tree $\hat{r}$ with less nodes than r_{full}. Furthermore the FSM closure of $\hat{r}$ ($\hat{r}'$) is a proper subtree of r' (see [19]). The algorithm will add more nodes to $\hat{r}'$ when a node in r' reached by the encoder is not present in $\hat{r}'$ during the decoding procedure.

FSM closure

The next step consists in the construction of the FSM closure of the tree built in the previous step using the same algorithm used in the encoding stage. During the following decoding procedure additional nodes will be added to this tree if needed.

Decoding

The decoding procedure starts at the root of the FSM closure in a similar fashion as the encoding. The encoded symbols are obtained from the arithmetic decoder using the statistics stored on the current node of the tree (via the *Origin* pointer). When a symbol is decoded the node statistics are updated and the FSM is used to move to the next decoding state. In case of decoding an escape the algorithm moves to the parent of this node searching again for the symbol to decode. As in each decoding step the statistics on the decoder tree are the same as the ones stored during the encoding phase the decoder is able to move up to the same state whose statistics where used to encode the current symbol.

SEE is used in the same way as in the encoding procedure. A context is found using data already known by the decoder in case the conditions for using SEE are met and the two values obtained from this context are used to decide if an escape was encoded or not. The SEE context is then updated in the same manner done by the encoder.

In case a walk up the tree is needed, after decoding the correct symbol the algorithm moves down the tree again updating the statistics on the intermediate nodes. This step is not needed during the encoding phase as the symbol at hand is already known while traversing the tree and the update is done during this initial walk. After this, the algorithm follows the FSM transition for the decoded symbol, creates new nodes if needed, and the procedure starts again until all symbols are decoded (the number of symbols in the original string is passed to the decoder along with the tree and the alphabet size). Essentially is the same process shown in algorithm 12.

The missing detail in the previous description is the process of adding new nodes to the tree during the decoding procedure. The idea is to add nodes to the tree in order to create the same state used in the encoder to process each symbol in case that state does not already exist in the tree. Together with this node additional nodes might be added in order to preserve the FSM property of the tree. More formally, the decoder uses a sequence of trees with the FSM property where each member of the sequence is obtained from the previous one by adding the node s_i together with the branching nodes resulting from this addition as well as the nodes needed to satisfy the FSM property where s_i is the state used to encode the symbol a_i in the encoder tree. The creation of this node starts from

the node $\hat{s}_i$ which is the node obtained by traversing the original FSM transition on the tree before adding any nodes.

This algorithm is presented with the corresponding proofs in [19] and will not be described here.

As was mentioned before, there are some issues related with the resetting of statistics done by the encoder. The problem is that, as the decoder tree is built incrementally as the symbols are decoded, it is possible that it has less nodes than the tree that the encoder uses. Considering that the decoding algorithm only guarantees that the initial state (i.e. the state before escaping) used in the encoder tree is present in the decoder tree it may happen that the upwards search for a node to encode a symbol not found in the initial state traverses more nodes in the encoder than the ones traversed by the decoder. In general this is not a problem as the additional nodes traversed by the encoder by definition where not used to encode any symbols (otherwise they would also be present in the decoder tree) and as such they will only contain escapes from the leaf where symbols were processed and so they will have the same number of symbols as the leaf and will not be used to encode a symbol that is not present on the leaf.

Resetting statistics changes all this as it is now possible that a symbol which occurred on a leaf ends up disappearing during the statistics reset but it is not affected in the node's parent as this has less symbols and is not reseted. In this way it can happen that the encoder processes a symbol in a state that is not present in the decoder tree breaking the whole process.

To solve this, when a symbol disappears during node resets it is also forced to disappear in the ancestors of this node that do not have symbols that occurred more than once. This forces the new symbols to be coded in some other node upwards which has another leaf as a descendant. This node has to exist in the decoder tree because the decoder only removed leaves and nodes with outgoing degree equal to one from the tree sent by the encoder. This is done both in the encoder and the decoder in case the state which has only escapes also exists in the decoder tree (remember that only the leaves were removed).

5.3 Algorithm implementation details

Next we will describe some details of the implementation of the semi-predictive context algorithm developed in this thesis.

5.3.1 Encoding

Here we will assume that the algorithm writes its output to a stream that can be connected to file or a network socket.

Initialization

First of all, the encoder needs to read the file to compress. This could not be accomplished in the traditional way by copying a portion of the file to a memory buffer, processing that buffer and then reading another one because we need random access to the whole file in order to build the suffix tree.

To solve this issue the complete file is mapped to memory using the *mmap* system call in UNIX systems or the *CreateFileMapping* and *MapViewOfFile* system calls in Windows systems. These system calls copy the whole file to the virtual address space of the calling process allowing it to access the file as if it was all available in RAM. In fact, the whole file does not need to be present in memory at the same time as the operating system can map the needed parts of the file on demand into the virtual memory of the process as well as remove part of this mapping if more RAM memory is required for other purposes.

An additional advantage of using a memory mapped file instead of using partial memory buffers is that there is no need for an additional copy of the file data to a user space buffer every time a new read operation is done. Moreover, reading the memory mapped file does not incur any system call or context switch overhead assuming that the portion of the file being read is already mapped. A possible disadvantage could be that, in 32 bit systems, this can limit the maximum size of a file to compress. Considering that the whole virtual space of a 32 bit process is 4 GiB it is not advised to try to compress files larger than 2 GiB.

The next task is to write to the output the alphabet of the input string. This could be accomplished by writing all the 8 bit symbols that occur on the input file but in order to save some space we do this

in a different way. First the total number of symbols is written and then, depending on this value, the alphabet or the complement of the alphabet is written. It is to say: if the alphabet is smaller than 128 all the symbols of the alphabet are written, otherwise only the symbols that do not belong to the alphabet are written. Obviously, if the alphabet size is 256 no symbols are written at all.

Also, the symbols themselves are encoded with an arithmetic coder in this way:

- The symbols are encoded in descending order starting with the ones with larger binary values.
- Each symbol is considered to have equal probability considering only the symbols that remain to be sent. So, the first symbol to send will have probability $\frac{1}{256}$. If, say, this symbol is 128 the next symbol will have probability $\frac{1}{128}$ and so on.

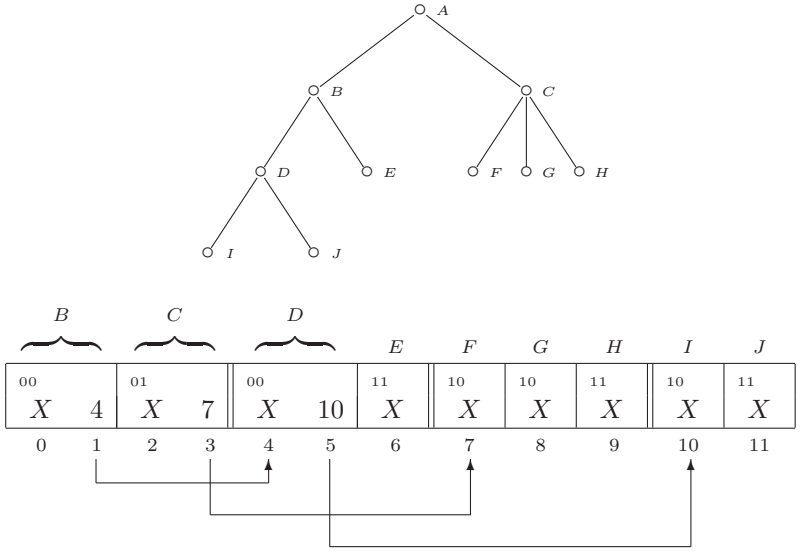
For example, if we need to encode the symbols with binary values 234, 122, 89 and 23 the probability used for each of them with the arithmetic encoder will be respectively $\frac{1}{256}$, $\frac{1}{234}$, $\frac{1}{122}$ and $\frac{1}{89}$.

Suffix tree structure and construction

The representation of the suffix tree used in the *wotd* algorithm is a simple array of integers. Each branching node requires two integers and the leaves use only one. The second integer on branching nodes is a reference to the position on the array where its first child is stored and the rest of the siblings are located sequentially in the array so that they do not need additional pointers. There is a flag that signals that a node is the last child which is used to indicate where the section of adjacent children nodes finishes. The two most significant bits of the first integer of every node are used as flags to signal if the node is a leaf and if the node is the last child of its parent node.

Let u be the string over which the suffix tree is being built. The first integer in branching nodes and the only one for leaves is a pointer to u used to indicate the section of u where the label for each node occurs for the first time. Formally the labels are associated with the edges and not the nodes but in this case they are stored in each edge's "destination" node.

Next, we present an example of a tree represented with this array structure leaving out the labels that will be discussed with more detail below (the pointers to the labels are marked with Xs). The nodes of the tree are marked with letters that are shown over the array elements that model each node in the array below. The two superscript bits represent the leaf and last child labels respectively. The arrows below make clear where pointers to the first children go in the array. It can be seen that the array representation has no root because the first elements in the array are assumed to be the children of the root node.



In general two references to u are needed to represent the labels of the tree nodes: one marking the beginning of the label and the other marking the end. In this array representation the leaves use only one integer to signal the position in the source string of the leftmost character of the edge label.

The rightmost character is not needed because for all leaves this is implicitly the last character of the string (\$).

On the other hand, it turns out that for branching nodes it is also sufficient to store only the left pointer of the node label. This is achieved by ordering the node's children so that the first child is the one whose edge label left pointer is the leftmost in u (has the smallest index). In this way, having a link from the parent to the first child, it is possible to find the right pointer of the parent label which is the left pointer of the first child minus one. We show an example of this below.

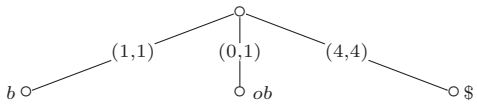
We will construct a context tree for the string $u = bobo$ so we will build a suffix tree for $\overline{u}\$$ ($obob\$$). As the algorithm determines the node's labels using pointers to the input string we will make explicit the ordinal number of each symbol of $\overline{u}\$$ that will be used subsequently:

o	b	o	b	\$
0	1	2	3	4

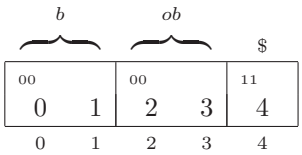
The first step of the algorithm (as shown in page 38) is to sort all the suffixes of $\overline{u}\$$ lexicographically[2] by the first symbol. The algorithm uses an array of suffixes ordered by decreasing length (called *suffixes*) to implement this so we also give the index of every suffix in the array. The horizontal lines divide the zones on the array where lay the suffixes that start with the same symbol.

0	bob\$
1	b\$
2	obob\$
3	ob\$
4	\$

Here we determine that the root of the tree has three children. Two of them are branching nodes and the other is a leaf. The *lcp* of the two branching nodes is "b" and "ob" respectively. Here we show the tree that is built at this stage where the numbers on the edges indicate the left and right pointers to the input string that signal the label of each edge. The label itself is also explicitly written on the nodes.



In the array structure this tree would be represented as:



The elements for the first two branching nodes contain the location in the sorted array of suffixes of the suffixes that start with the branching node prefix. These nodes will be processed recursively during the DFS walk of the tree and then their children will be created and the node values will change. The nodes that contain references to the *suffixes* array are called *unevaluated* nodes and are marked as such. The leaf node is already finished and it only contains the position of the first symbol of the edge label (the last one being \$ implicitly).

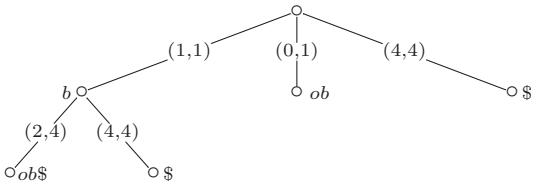
The next step is to evaluate the unevaluated node corresponding to the *lcp* "b". The suffixes 0 and 1 of the *suffixes* array will be sorted by their second symbol leaving out the *lcp* that is "b". We also need the symbols to be stored in the order of their appearance in the *suffixes* array. This is because, as already mentioned, the first child of the node must be the one whose label left pointer is leftmost in u. As the array of suffixes is originally ordered by suffix length and the counting sort is stable[3], this is the leftmost suffix in the interval. In our case the suffixes are already correctly sorted so we get:

[2] Any order will work as long as the suffixes that start with the same symbol are grouped together

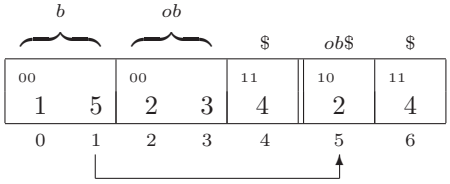
[3] Stable sorting algorithms maintain the relative order of records with equal keys

0 (b)ob$
1 (b)$

This means we have to add two leaves to this node. The tree now will be like:



And now the array representation of this tree is:

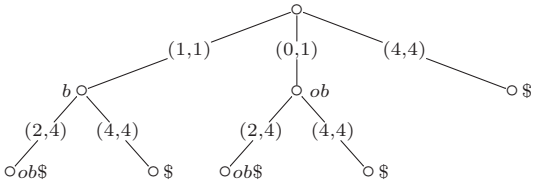


We can see that the two elements that represent the node "b" have changed its contents. The first number is a reference to the position on the $\overline{u}$\$ string where this edge label starts. In this case the label is "b" and starts in the position 1 of $\overline{u}$\$. This position can be obtained easily as it is the position in $\overline{u}$\$ of the first symbol of the first suffix in the group. The end of the label can be obtained by looking at the first child pointer (in this case 2) and subtracting one. This implies that the edge string is (1,1), so it is the symbol "b". The second element is a reference of the position of the first child in the array. In this case is the position 5 which is indicated by the arrow below. There are no recursive calls in this case as the two children of the node are leaves.

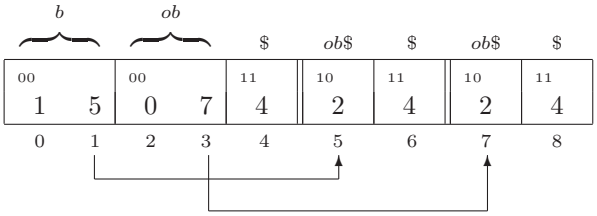
Then the next (and last) step is to evaluate the unevaluated node whose *lcp* is "ob". Sorting the subsection $[2, 3]$ of the array of suffixes by the third symbol (as "ob" is the common prefix) we get:

2 (ob)ob$
3 (ob)$

So again we have two leaves leaving the tree as:



Adding this two leaves to our array representation gives:



As before the contents of the two elements of this node have changed. The first one is now 0 (the index in $\overline{u}$\$ of the suffix *obob*\$) representing that this edge label is the subsection $[0, 1]$ of $\overline{u}$\$ (ie. "ob"). The second value is 7 which is the position on the array where the children of this node are

located. The label of the edges leading to the two leaves are exactly the same as the previous case so the array contents are the same.

To summarize, the construction of the suffix tree proceeds evaluating nodes starting from the root and continues processing nodes recursively downwards into the subtrees. To evaluate an unevaluated node s with label v the algorithm needs to have access to all the suffixes that start with v. In order to achieve this the algorithm uses the *suffixes* array that contains pointers to all suffixes of $\overline{u}\$$. For each unevaluated node its two integers point to an interval of the *suffixes* array that contains the suffixes that start with the label of this node ordered by descending suffix length. To distinguish evaluated and unevaluated nodes there is an additional flag (the third most significant bit of each integer).

The edges outgoing from s are obtained using a counting sort by the first character of each suffix in the interval of *suffixes*. These suffixes are ordered grouping equal symbols together so that the leftmost suffix remains to the left. Each character a with a count greater than zero corresponds to an edge whose label starts with a outgoing from s. The complete label for the edge is obtained calculating the least common prefix (*lcp*) of all the suffixes in the group. If there is only one suffix in the group the edge leads to a leaf and the *lcp* is the full suffix. All the children nodes are stored consecutively in the array reserving two elements for branching nodes and one for leaves.

All the new nodes that are not leaves are, by definition, unevaluated and so the pointers to the corresponding range in *suffixes* must be stored. These pointers are obtained from the sorted section of the *suffixes* array incrementing every suffix by the length of the corresponding *lcp*. The last step consists of changing the two elements of the node s storing the left pointer of the node label in the first one and the pointer to its first child in the second. Finally the unevaluated flag is removed.

In our implementation this tree structure was modified in order to store the counts of the occurrences of symbols on branching nodes. In order to achieve this an additional array element is used for branching nodes. This extra integer is used to store a pointer to the counts on the node.

Pruning

The tree is pruned during its construction simply by computing $\Omega'(s)$ after a subtree becomes completely evaluated and storing the result on the root of the subtree. As the tree is built in a DFS fashion using a stack we can return to each node after all children subtrees are evaluated. If the node needs to be pruned the children are deleted and the node becomes a leaf. There is no extra work in making the node atomic as the only value stored in the node is the first symbol of its label.

As the tree is already pruned at the end of the last step, the only remaining step is to write the tree to the output stream. The tree is traversed in-order and completed to create a full atomic tree during this traversal. Each node is encoded and written when it is reached in the traversal.

Encoding

Next we will comment on some of the changes to the algorithm done for tuning purposes during testing.

To implement the periodic reset of statistics we determined that binary files required more frequent resets than text files. So we use the number of symbols in the file as a heuristic method to differentiate the two kind of files. If we let A be the number of symbols in the file and $c_s(a)$ be the counts of symbol a in state s the reset is done when:

$$\begin{cases} c_s(a) > 450 & \text{if } A \leq 100 \\ c_s(a) > 200 & \text{otherwise} \end{cases}$$

When using SEE, we obtain two values called SEE_1 and SEE_2 and use them to estimate the escape probability. The escape probability will be SEE_1/SEE_2 and the probability of not escaping is $(SEE_2 - SEE_1)/SEE_2$

In this way, when a SEE context is used, the arithmetic coder can be used two times: the first one to encode if an escape is emitted or not (one bit) using the probabilities shown before, and the second to encode the found character in case there was no escape. This character is encoded using only the probability of the characters not masked in the current state because the decoder already knows that the symbol to encode is effectively found in the current state.

This SEE technique is only used when:

- There are occurrences of non masked characters in the current state (otherwise the state is ignored by the encoder and the decoder)
- The total number of symbols in this state is less than 30 for text files or 128 for data files

After each use of the SEE context the statistics of the context are updated to reflect the fact that the searched symbol was found or not. The context is updated according to the following criteria:

- If the symbol was found the second value of the context is incremented by 17 for text files and 16 for data files.
- If an escape was emitted the first value is incremented by 17 and the second by 18

These increments are large in order to forget old statistics rapidly and adapt to new ones. The two values on each SEE context are halved when any of them reach an upper limit which is different for data and text files in a similar fashion as the statics resets in the main algorithm.

This fine tuning was performed using different files than the ones used for comparison in section 6.

5.3.2 Decoder

Tree building

The tree structure implementation used in the decoding stage is different from the one used while encoding because the decoder obviously does not have access to the original string. The labels of the edges in the encoder tree are represented using pointers to this string. In this case this is not possible and the simple solution is to have a pointer to the label on every edge.

As the decoding algorithm sometimes needs access to the complete context of a node (ie. to all the symbols in the edges on a path from the root to the node) we store this full string on every edge with pointers indicating the suffix that corresponds to the label of this particular edge.

5.4 Algorithm execution analysis

In order to evaluate the algorithm behavior when encoding and decoding files it was run using a profiling tool (*Valgrind*). This tool allows us to obtain a graphic representation showing the code areas where the most time is spent.

5.4.1 Encoding

In this case clearly the three most expensive activities are building the suffix tree (which includes pruning), creating the FSM closure and encoding the input file using this tree. There is a notorious difference in the relative time needed for these three activities between text and data files. It is important to consider also that data files take more time than text files as will be show in section 6.

In the case of data files the encoding phase tends to be the longest taking about 40% of the time. In this case the FSM closure takes about 25% of the total time and the suffix tree construction takes about 15% to 20% of the total time the proportion being larger with larger files.

On the other hand in text files the longest phase is the suffix tree construction which takes from 50% to 65% also being larger with larger files. The coding phase takes from 25% to 30% and the FSM construction takes about 5% to 10%. In this case the other tasks are relatively shorter as the tree construction phase gets larger.

This is explained mainly in terms of the encoding stage which takes more time on binary files. It happens that binary files have a larger number of escape symbols than text files. The reason for this is that binary files have a larger alphabet, usually near 256, and this produces trees with a larger number of leaves (i.e. states). This causes the algorithm to take more time to capture relevant statistics. Moreover, as was pointed out before, binary files usually benefit from more frequent statistics resets. This also causes an extra number of escapes as the algorithm learns the new statistics.

This difference can also be appreciated in section 6 where it is shown that encoding binary files with resets is approximately 10% slower than not using them. There is a trade-off in terms of the coded file size.

Considering the tree building stage, as text files usually have taller trees, it normally takes more time for text files than for binary ones.

5.4.2 Decoding

The decoding procedure is simpler as it only requires to read the tree from the file in order to start decoding. Here the decoding procedure takes about 90% of the time in text files and about 95% in data files.

Considering this, it is interesting to study the time spent in the different steps within this decoding stage. As was mentioned before, binary files tend to have a larger number of escapes which means the decoding procedure is expected to take longer. In fact, if we consider only the time spent in the decoding procedure alone[4] and ignore the time consumed by the procedures invoked from there, binary files take about 70% of the time here whereas text files spend about 50%. This is approximately constant across different input file sizes.

It is also important to note that the decoding procedure starts by removing all leaves of the full tree and has to add new ones as the algorithm proceeds. As binary files have larger alphabets, the tree has more leaves and then more work is needed in order to add all these nodes.

[4]This includes the time used by the function which adds new nodes to the tree. As these tests where run with compiler optimization enabled this function is inlined into the main decoding procedure as it is only called from there. In fact, it was defined as a separate function in the source code for clarity purposes only.

Chapter 6

Experimental results

To evaluate the relative performance of the different algorithms several tests were performed using files of different types and different sizes. The files where separated into text only files and binary data files because, as was noted before and will be seen from the experimental results, they have different behavior during compression and decompression. We also separate the data considering the approximate size of the files. The data is averaged for files of the same type and size. The files used for comparison were:

bible.txt A section of the bible in english (1012 KiB)

book2 Text in english (from Calgary corpus) (600 KiB)

circuitos.doc Microsoft Office file (52 KiB)

compat.xml Docbook file in english (100 KiB)

cookies.sqlite SQLite database (1024 KiB)

font.ttf TrueType font (608 KiB)

libfltk_images.so.1.1 ELF 32-bit shared object (52 KiB)

librtmp.so.0 ELF 32-bit shared object (104 KiB)

object.o ELF 64-bit object file (596 KiB)

overlay_spanish.txt Translation file from english to spanish (48 KiB)

random.c C code file (52 kiB)

smsserver.log Text log file (596 KiB)

steamui_spanish.txt Translation file from english to spanish (1008 KiB)

super.c C code file(104 KiB)

timezones.png Image file (100 KiB)

user32.dll.so ELF 32-bit shared object (1036 KiB)

First we compare the final implementation of our algorithm with itself showing a base version and then adding optimizations individually in order to show the improvements contributed by each one. Then we compare our algorithm with other algorithms. The compression and decompression times were evaluated separately for each algorithm

All the algorithms were run over the same files in the same hardware three different times and the processing times were averaged. The five algorithms compared where:

bzip2 1.0.5 A BWT based algorithm by Julian Seward

ctw 0.1 A CTW implementation by Erik Franken and Marcel Peeters

ppmd variant I ver. 1 A PPM implementation by Dmitry Shkarin

zpaq 1.02 A context mixing algorithm by Matt Mahoney

context The algorithm presented in this thesis

The hardware used for these tests was:

- AMD Phenom II X3 720 Processor @ 2.80 GHz
- 4 GiB RAM
- Linux kernel version 3.10.7

In the following tables all times are given in seconds and compressed file sizes in bytes.

6.1 Optimizations

Here we compare different versions of our algorithm with the following features:

- Base version
- Builds the tree using the Ukkonen algorithm [36] and prunes it afterwards in a separate step.
- Uses several sub-trees. The input is partitioned in five parts in this case and each one of them is processed using a different tree.
- Resets count statistics (some resets are always needed in order to avoid number representation overflow but they are far more frequent than in the base implementation)
- Uses SEE.
- Count resets and SEE optimizations combined (final version)

6.1.1 Text files

In table 6.1 we show the compression times and sizes for text files of different sizes. When the Ukkonen algorithm is used to build the suffix tree the times are longer in all cases. The compressed file size is the same as the final algorithm as the pruned tree is the same.

Here we see that adding resets and SEE results as expected in slower execution time but shorter compressed files. SEE is more expensive than resets but gives better results.

The division in subtrees results in longer execution time and worse compression rate. In this case the statistics are not being copied between trees. This is better for data files as shown below although not really useful.

Considering decompression time we can see that it is considerably faster than compression in all cases as there is no need to build and prune the context tree which is a costly operation. The relative time differences remain the same as for compression except that in some cases the final algorithm is slower than the partition in subtrees. The difference is that the smaller trees are processed faster during decompression as there are less nodes to add to them. As the base algorithm times are shorter, the use of SEE adds a greater proportion of time during decompression. Also we see that the use of the Ukkonen algorithm makes no difference in the decoding stage as expected.

6.1.2 Binary files

Table 6.2 shows the times and sizes for binary files of different sizes. In general times are longer than in the text case and the use of the Ukkonen algorithm also shows longer compression times.

Here we also see that resetting statistics is slower than in the text case as they are reseted more frequently in binary files. On larger files, resets are better than SEE although they are more expensive in all files. Also the cost in time is more in proportion during decompression as the cost of these optimizations is mostly symmetrical and the rest of the algorithm is faster.

We also see that the partition in subtrees obtains better results than the base algorithm in some cases. It is not shown here but, even in this case, adding the partition in subtrees to the final algorithm

Input size	Algorithm	Comp. time (seconds)	Comp. size (bytes)	Decomp. time (seconds)
50 KiB	Base	0,06	10585	0,03
	Ukkonen	0,09	10585	0,03
	Sub-trees	0,07	12514	0,03
	Reset	0,06	10569	0,03
	SEE	0,06	10545	0,04
	Final	0,07	10528	0,04
100 KiB	Base	0,11	21101	0,06
	Ukkonen	0,20	21101	0,06
	Sub-trees	0,14	26836	0,07
	Reset	0,11	21087	0,07
	SEE	0,12	20985	0,07
	Final	0,12	20970	0,07
600 KiB	Base	0,52	91023	0,29
	Ukkonen	1,12	91023	0,29
	Sub-trees	0,63	114266	0,33
	Reset	0,52	90049	0,29
	SEE	0,54	89905	0,31
	Final	0,55	88932	0,32
1 MiB	Base	0,91	145567	0,54
	Ukkonen	2,18	145567	0,54
	Sub-trees	0,96	166674	0,57
	Reset	0,91	145006	0,55
	SEE	0,95	144719	0,58
	Final	0,96	144158	0,59

Table 6.1: Optimization in text files

does not produce better results. The cost of relearning the statistics for a new tree outweighs the benefits of having a tree more precisely fitted to a particular section of the file. We also tried copying statistics between nodes of the partitioned trees but in practice it is not very different than using resets. As always it is better for data files as they benefit more from this kind of resets.

Finally, considering decompression times the relative times are the same as in the case of compression but, as in the previous case, they are shorter.

6.2 Other algorithms

The five algorithms where compared in terms of the size of the compressed files and the average time needed to process them, both during compression and decompression.

6.2.1 Text files

Table 6.3 shows that in compression and decompression time BZIP is the fastest algorithm followed by PPM, Context, CTW and ZPAQ. This is expected as BZIP is much simpler than the other algorithms and also is the only one that does not use floating point arithmetic at all. The operations in BZIP are mostly string manipulations which are much faster than probability estimations, pruning, context weighting or arithmetic coding. Also BZIP and Context have the property of having slower decompression times as their compression and decompression algorithms are not symmetrical. In general ZPAQ has the best compression ratio followed by PPM, Context, CTW and BZIP. CTW and Context are very similar on average in this case.

6.2.2 Binary files

When data files are compressed (table 6.4), all algorithms except BZIP take more time than the time they took to compress text files of similar sizes. PPM and Context take twice the time or more. In the case of Context it was shown before that this is caused by the larger number of escapes in data

Input size	Algorithm	Comp. time (seconds)	Comp. size (bytes)	Decomp. time (seconds)
50 KiB	Base	0,14	32987	0,09
	Ukkonen	0,23	32987	0,09
	Sub-trees	0,12	32747	0,06
	Reset	0,17	32629	0,13
	SEE	0,15	32238	0,09
	Final	0,18	31589	0,12
100 KiB	Base	0,34	56477	0,18
	Ukkonen	0,56	56477	0,18
	Sub-trees	0,35	58195	0,18
	Reset	0,38	56713	0,24
	SEE	0,35	55650	0,19
	Final	0,40	55499	0,23
600 KiB	Base	2,17	233213	1,16
	Ukkonen	3,62	233213	1,16
	Sub-trees	2,03	231254	1,07
	Reset	2,40	222931	1,45
	SEE	2,23	228731	1,22
	Final	2,48	217059	1,44
1 MiB	Base	2,41	283755	1,31
	Ukkonen	4,58	283755	1,31
	Sub-trees	2,63	289587	1,36
	Reset	2,61	274809	1,56
	SEE	2,48	279867	1,38
	Final	2,72	270002	1,59

Table 6.2: Optimization in binary files

files and also because of the fact that the tree built to model the data has many more nodes that the ones used with text files of similar sizes. This is not caused by the tree being higher but mostly because the eight-bit alphabet size in binary files is always near 256 while in text files it is usually below 100. As CTW and PPM also build trees to model the data this probably can explain the difference that is not seen in BZIP. Considering the compressed files size the order is similar to the one seen on text files.

In the case of decompression time, the same pattern seen in previous tables emerges.

6.3 Standard test corpora

These algorithms where also tested using the standard Canterbury and Calgary corpora. These corpora are developed specifically to test compression algorithms and are composed of a set of test files representative of different common file types. To compare with the previous tables, the Calgary corpus is composed by 12 files where the two largest are about 500 KiB and 8 of them are 100 KiB or smaller. The Canterbury corpus consists of 11 files where 4 of them are larger than 400 KiB (the largest is almost 1 MiB) and five of them are smaller than 100 KiB. On average Canterbury corpus files are about 32 KiB larger than Calgary's. The results for both of these tests averaged over all files in each set can be seen in tables 6.5 and 6.6. The results are similar to the ones observed before. CTW is better than Context as both corpora contain small files mostly. For comparison we add the result of processing the previous 16 files tested considered as a corpus (table 6.7).

Comparing the three tables it is clear that Context does better when compressing larger files as the difference to PPMD in bits per byte is reduced from 10% in Calgary to 7% in Canterbury to 6% in our test files.

Input size	Algorithm	Comp. Time (seconds)	Comp. size (bytes)	Decomp. time (seconds)
50 KiB	BZIP	0,01	10478	0,00
	Context	0,07	10528	0,04
	PPM	0,02	9008	0,03
	CTW	0,19	10043	0,19
	ZPAQ	4,22	8752	1,77
100 KiB	BZIP	0,02	21027	0,00
	Context	0,12	20970	0,07
	PPM	0,04	18001	0,05
	CTW	0,38	20183	0,40
	ZPAQ	5,91	17005	3,48
600 KiB	BZIP	0,09	93327	0,02
	Context	0,55	88932	0,32
	PPM	0,17	81009	0,18
	CTW	2,44	89134	2,48
	ZPAQ	22,27	72243	20,15
1 MiB	BZIP	0,15	147644	0,04
	Context	0,96	144158	0,59
	PPM	0,29	124852	0,32
	CTW	4,13	153447	4,20
	ZPAQ	35,49	120093	33,65

Table 6.3: Other algorithms in text files

6.4 Conclusions

Considering the compression ratio and the time consumed in the compression procedure we can conclude that the best algorithm is PPM followed by BZIP which gets a slower compression ratio but it is much faster especially on binary files. Context is similar to CTW with small files but it is much faster (especially during decompression) and compresses better with larger files.

A possible explanation of the performance of PPM is that this is a highly optimized algorithm which uses many heuristic techniques to improve compression. On the other hand the implementation of Context cannot obtain so much benefit from this kind of heuristic techniques because of the fact that it works with a fixed tree previously determined. Of course this tree is supposed to be optimal but it is not possible to take into account all possible heuristic improvements in the cost formula used in the pruning stage. For this reason the heuristic improvements implemented within the algorithm produce little improvement in compression but do not compare with the benefits that can produce using a free form tree like in PPM.

Nevertheless the differences between Context and PPM are smaller with larger files in general and especially with data files. In our tests the difference in compression between PPM and Context in data files was always less than about 4%. To show this we present the results of our test files as a corpus considering only data files larger than 500 KiB (table 6.8).

Input size	Algorithm	Comp. Time (seconds)	Comp. size (bytes)	Decomp. time (seconds)
50 KiB	BZIP	0,01	32387	0,00
	Context	0,18	31589	0,12
	PPM	0,08	31085	0,10
	CTW	0,18	31373	0,18
	ZPAQ	4,38	29524	1,91
100 KiB	BZIP	0,02	57431	0,01
	Context	0,40	55499	0,23
	PPM	0,12	53194	0,14
	CTW	0,38	54982	0,38
	ZPAQ	6,03	50364	3,60
600 KiB	BZIP	0,07	231920	0,04
	Context	2,48	217059	1,44
	PPM	0,52	209966	0,60
	CTW	2,68	219273	2,73
	ZPAQ	22,52	163727	20,42
1 MiB	BZIP	0,10	285944	0,04
	Context	2,72	270002	1,59
	PPM	0,62	257992	0,71
	CTW	4,58	286724	4,63
	ZPAQ	36,40	207228	34,56

Table 6.4: Other algorithms in binary files

	Bits per byte	Compression time (seconds)	Decompression time (seconds)
BZIP	2,36	0,02	0,01
PPM	2,10	0,09	0,09
Context	2,32	0,25	0,15
CTW	2,23	0,83	0,84
ZPAQ	1,91	9,42	6,69

Table 6.5: Calgary corpus results

	Bits per byte	Compression time (seconds)	Decompression time (seconds)
BZIP	2,22	0,03	0,01
PPM	1,98	0,13	0,15
Context	2,12	0,24	0,16
CTW	2,03	1,07	1,08
ZPAQ	1,87	11,60	8,87

Table 6.6: Canterbury corpus results

	Bits per byte	Compression time (seconds)	Decompression time (seconds)
BZIP	2,53	0,06	0,02
PPM	2,30	0,24	0,27
Context	2,44	0,95	0,56
CTW	2,45	2,11	2,10
PAQ	2,09	18,69	15,85

Table 6.7: Tested files as a corpus results

	Bits per byte	Compression time (seconds)	Decompression time (seconds)
BZIP	2,58	0,09	0,04
PPM	2,33	0,57	0,65
Context	2,42	2,63	1,54
CTW	2,50	3,63	3,69
PAQ	1,38	31,25	28,50

Table 6.8: Tested files as a corpus results, only data files larger than 500 KiB

Chapter 7

Conclusions and future work

It was shown in this thesis that it is possible to develop an algorithm with sound theoretical properties and also reasonable compression rate on everyday files. There is still a difference between the algorithm developed here and the best performing algorithms in practice. Further work would be needed to shorten this breach.

The complete source code of the implementation of the algorithm defined here is available on the web on http://www.fing.edu.uy/~jmerlino/context thus it is possible that the community works to improve it. We note that some possible lines of work on this regard are:

- Work further on the idea of using subtrees instead of a single tree.
- Adapt other optimization techniques from other compression algorithms.
- Make the pruning formula reflect better all these possible code optimizations considering that it should be computed relatively fast for the algorithm to be competitive.

Appendix A

Counting sort

This is a sorting algorithm that is used in the construction of the suffix tree in chapter 5. It is very efficient when the range of the elements to sort is known and is not much larger than the length of the array to sort. In this case it is used to sort 8 bit symbols. The counting sort has a running time of $\Theta(n+k)$ where n is the length of the array to sort and k is the range of the values to sort.

As explained in [14, section 5.2] this method is based on the simple idea that the j^{th} element in the sorted sequence is greater than exactly $j-1$ of the other elements. Hence, if we can count how many elements are smaller than a particular one we know the position of this element in the sorted array.

Algorithm 13: Counting sort

input : String a^n to sort
output: Array C, $C[a_i]$ = position of the symbol a_i in the sorted string

```
1 for i ∈ A do
2 |   C[i] = 0
3 end
4 for i = a_1 to a_n do
5 |   C[i] = C[i] + 1
6 end
7 for i = A_2 to A_k do // A_i is the i-th symbol of the alphabet
8 |   C[i] = C[i] + C[i − 1]
9 end
```

The algorithm starts by creating an array C with the same size as the range of the elements to sort in such a way that it can be indexed by the values of these elements. This array will be zero-filled at the beginning of the algorithm (lines 1-3). The value on index i of this array is used to store the number of elements in the data to sort that have value i. These counts are accumulated iterating over the array to sort (n steps, lines 4-6).

Afterwards, the counting array C is iterated in increasing order of its keys and the counts are accumulated (lines 7-9). At this point the array C contains the position of each symbol in the sorted array. If there are repeated elements in the original data the value $C[i]$ will contain the position of the last symbol i in the array. The previous free positions have to be filled with this same symbol. This takes k steps to form the $n+k$ total algorithm time.

As an example lets see how the algorithm would sort the string *abracadabra*:

1. Initialize the count array: $C[a] = C[b] = C[c] = C[d] = C[r] = 0$
2. Count the number of occurrences of each element. The resulting array will be: $C[a] = 5$, $C[b] = 2$, $C[c] = 1$, $C[d] = 1$, $C[r] = 2$
3. After the accumulation step the array will now be: $C[a] = 5$, $C[b] = 7$, $C[c] = 8$, $C[d] = 9$, $C[r] = 11$

4. The sorted string including the position of each symbol is then:

a	a	a	a	a	b	b	c	d	r	r
1	2	3	4	5	6	7	8	9	10	11

A possible variant of this algorithm is to skip the accumulation step and iterate over the array C instead. Each symbol is stored in the sorted array as many times as indicated by the count in C. This is useful if a new ordered array with the original elements is needed instead of an array with pointers to the original one as is the result of the former algorithm.

As this algorithm uses counts to order the array it is not a comparison sort so the $\Omega(n \log(n))$ lower bound is not applicable in this case.

Appendix B

Pruning formula details

For a given state s if we let c_s be the number of symbols that occur in s in the entire input sequence, let $c_s(a)$ be the number of times the symbol $a \in A$ occurs in s as recorded by the counter, A_s be the subset of A composed by the symbols that occur in s and thus $|A_s|$ is the number of different symbols that occur in s. Then the code length estimation for this state is given by:

$$\kappa(a^n, s) = \sum_{a \in A_s} \sum_{i=1}^{c_s(a)-1} (-\log p_{s,i}(a))$$

where

$$p_{s,i}(a) = \frac{i - \frac{1}{2}}{c_{s,i}(a)}.$$

and $c_{s,i}(a)$ is the number of symbols that occurred in s at the time symbol a occurs for the i^{th} time.

This can be written as:

$$\kappa(a^n, s) = -\log \frac{\prod_{a \in A_s} \prod_{i=1}^{c_s(a)-1} (i - \frac{1}{2})}{(c_s - 1)!} \tag{B.1}$$

Here we can see that by estimating probabilities using PPMD from [12] the denominator of (B.1) is given by $(c_s - 1)!$ as $c_{s,i}(a)$ increases by one in every step. It is not $c_s!$ because the state makes no prediction the first time a symbol is seen there since $c_{s,i}(a) = 0$ in this case.

Next, considering the numerator of (B.1) we compute separately the number of occurrences of every symbol when it is predicted and the number of occurrences of the escape symbol.

For each symbol the number of occurences when predicted is given by:

$$[2(c_s(a) - 1) - 1]!! \, 2^{-(c_s(a)-1)}$$

as each symbol count increases by $\frac{1}{2}$ the first time it occurs (an escape is generated) and then the count increases by $\frac{2}{2}$ for each occurrence. For example, given a symbol that occurred five times, the counts for each of the four successful predictions of the symbol would be: $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}$.

This term can also be expressed using the gamma function as:

$$\Gamma\left(c_s(a) - \frac{1}{2}\right) \pi^{-\frac{1}{2}}$$

And considering the product over all of the symbols occurring in the state we get:

$$\prod_{a \in A_s} \left[\Gamma\left(c_s(a) - \frac{1}{2}\right)\right] \pi^{-\frac{|A_s|}{2}}$$

Finally, considering the number of escapes we have that the count increases by $\frac{1}{2}$ for each new escape. The number of escapes is $|A_s| - 1$ as one is produced for each new symbol except for the first one as no prediction is made in this case. Then we have that the product of the number of escapes is:

$$\left(\frac{1}{2}\right)^{|A_s|-1}(|A_s|-1)!$$

Mixing all these terms we get the final formula as:

$$\kappa(a^n,s) = -\log \frac{\prod_{a\in A_s}\left[\Gamma\left(c_s(a)-\frac{1}{2}\right)\right]\pi^{-\frac{|A(s)|}{2}}\left(\frac{1}{2}\right)^{|A_s|-1}(|A_s|-1)!}{(c_s-1)!}$$

Bibliography

[1] N. Abramson, *Information theory and coding*, New York: McGraw-Hill, 1963, pp. 61-62

[2] T. Bell, I. H. Witten, J. G. Cleary, "Modelling for Text Compression", *ACM Computing Surveys*, vol. 21, no. 4, pp. 557-591, 1989

[3] H. Blasbalg, R. Van Blerkom, "Message Compression", *IRE Transactions on Space Electronics and Telemetry*, vol. SET-8, issue 3, pp. 228-238, 1962

[4] J.L. Bentley, D.D. Sleator, R.E. Tarjan, V.K. Wei, "A locally adptive data compression scheme", *Communications of the ACM*, Vol 29, No. 4, pp-320-330, April 1986

[5] J. Bentley, R. Sedgewick, "Fast algorithms for sorting and searching strings", *Proceedings of the 8th ACM-SIAM Symposium on discrete algorithms*, pp. 360-369, 1997

[6] C. Bloom, "Solving the problems of context modelling", http://www.cbloom.com/papers/index.html

[7] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm", *Digital SRC Research Report 124*, May 1994

[8] J. G. Cleary, I. H. Witten, "Data compression using adaptive coding and partial string matching", *IEEE Trans. Commun.* COM-38, pp. 1917-1921, 1984

[9] M. Effros, "PPM Performace with BWT Complexity: A Fast and Effective Data Compression Algorithm", *Proceedings of the IEEE Data Compression Conference 2000*, March 2000

[10] M. Effros, K. Visweswariah, S. R. Kulkarni, S. Verdú, "Universal Lossless Source Coding With the Burrows Wheeler Transform", *IEEE Transactions on Information Theory*, 48(5), pp. 1061-1081, May 2002

[11] R. Giegerich, S. Kurtz, and J. Stoye, "Efficient implemetation of lazy suffix trees", `Proceedings of the 3rd Workshop on Algorithm Engineering`, number 1668 pp. 30-42, 1999

[12] P.G. Howard, "The design and analysis of efficient lossless data compression systems", Technical Report CS9328, Brown University Providence, Rhode Island, 1993.

[13] F. Jelinek, *Probabilistic information theory*, New York: McGraw-Hill, 1968, pp. 476-489

[14] D. E. Knuth, "The Art of Computer Programming", Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0.

[15] R. Krichevsky, "Laplace's law of succession and universal encoding", *IEEE Trans. Inform. Theory*, vol. 44, pp. 296-303, 1988

[16] R. Krichevsky, V. Trofimov, "The performance of universal encoding", *IEEE Trans. Inform. Theory*, vol. 27, pp. 199-207, 1981

[17] P. Laplace, "Philosophical Essays on Probabilities", New York: Springer-Verlag, 1995 (A. I. Dale, transl. from ed. 5, 1825)

[18] M. Mahoney, "Adaptative Weighting of Context Models for Lossless Data Compression", *Technical Report CS-2005-16, Florida Tech., USA*, 2005

[19] A. Martin, G. Seroussi, M. J. Weinberger, "Linear time universal encoding of tree sources", *Hewlett-Packard Laboratories Technical Report*, May 2003

[20] A. Martin, G. Seroussi, M. J. Weinberger, "Linear time universal coding and time reversal of tree sources via FSM closure", *IEEE Trans. Inform. Theory*, vol. 50, pp. 1442-1468, 2004

[21] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., September 1952, pp 1098-1102

[22] G. N. N. Martin, "Range encoding: an algorithm for removing redundancy from a digitized message", *Video & Data Recording Conference*, Southampton, UK, July 1979

[23] E. M. McCreight, "A space-economical suffix tree construction algorithm", *Journal of the ACM*, 23(2) pp. 262-272, 1976

[24] N. Merhav and M. Feder, "Universal Prediction", *IEEE Trans. Inform. Theory*, vol. IT-44, pp. 2124-2147, October 1998.

[25] D. R. Morrison, "Patricia - practical algorithm to retireve information coded in alphanimeric", *Journal of the ACM*, vol. 15, no. 4, pp. 514-534, 1968

[26] R. Pasco, "Source coding algorithms for fast data compression", Ph.D. dissertation, Stanford Univ., Stanford CA, 1976

[27] J. Rissanen, "Generalized Kraft inequality and arithmetic coding", *IBM J. Res. Devel.*, vol. 20, p. 198, 1976

[28] J. Rissanen, "A universal data compression system", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 5, pp. 656-664, 1983

[29] J. Rissanen, "Stochastic complexity and modelling", *Annals of Statistics*, vol. 14, pp. 1080-1100, September 1986

[30] J. Rissanen, "Universal coding, information, prediction and estimation", *IEEE Trans. Inform. Theory*, vol. IT-30, pp. 629-636, July 1984

[31] B. Ryabko, "Twice universal coding", *Problems of Information Transmission*, vol. 20, pp. 173-177, July/September 1984

[32] J.Seward, "On the performance of BWT sorting algorithms", *Proceedings of the Data Compression Conference*, pp. 173-182, March 2000

[33] J. Seward, "Space-time Tradeoffs in the Inverse B-W Transform", *Proceedings of the IEEE Data Compression Conference 2001*, March 2001

[34] C. E. Shannon, "A Mathematical Theory of Communication", *Bell Syst. Thech J*, pp. 398-403, July 1948.

[35] D. Skarin, "PPM: one step to practicality", *Proceedings of the IEEE Data Compression Conference 2002*, March 2002

[36] E. Ukkonen, "On-line construction of suffix trees", *Algorithmica*, 14(3) pp. 249-260, 1995

[37] M. J. Weinberger, J. Rissanen, M. Feder, "A universal finite memory source", *IEEE Trans. Inform. Theory*, vol. IT-41, pp. 643-652, May 1995

[38] P. Weiner "Linear pattern matching algorithm", *14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 111, 1973

[39] F. M. J. Willems, Y. M. Shtarkov and T. J. Tjalkens, "The context-tree weighting method: Basic properties", *IEEE Trans. Inform. Theory*, vol. IT-41, pp. 653-664, May 1995.

[40] F. M. J. Willems, Y. M. Shtarkov and T. J. Tjalkens, "Reflections on the Prize Paper: The context-tree weighting method: Basic properties", *IEEE Inf. Theory Soc. Newsletter*, vol. 47, pp. 20, March 1997.

[41] I. H. Witten, R. M, Neal and J. G. Cleary, "Arithmetic Coding for Data Compression", *Communications of the ACM*, vol 30, pp 520-540, June 1987

[42] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, 23(3), pp. 337-343, May 1977

[43] http://www.ele.tue.nl/ctw

[44] http://mattmahoney.net/dc/zpaq.html