Complexity reduction of the context-tree weighting algorithm : a study for KPN Research

Complexity Reduction of the Context-Tree Weighting Algorithm: A Study for KPN Research

F.M.J. Willems
Tj. J. Tjalkens

Author address:

Information and Communication Theory Group,
Eindhoven University of Technology, Eindhoven

EIDMA report series
EIDMA-RS.97.01

F.M.J. Willems
Tj. J. Tjalkens

Complexity Reduction of the Context-Tree Weighting Algorithm:
A Study for KPN Research

Eindhoven: Euler Institute for Discrete Mathematics and its Applications
ISSN: 1384-6353
ISBN: 90-75332-03-3
NUGI 811/832
Subject headings: information theory, data compression

Contents

ABSTRACT. This report contains the results of a study that was performed by the Information and Communication Theory Group of Eindhoven University for KPN Research in Leidschendam, The Netherlands. The objective of this study was to improve upon the implementation of the Context-Tree Weighting (CTW) algorithm that was produced in previous work for KPN Research. The improvements that were desired were reduction of the storage compexity and a higher processing speed. This report shows that the storage complexity can be reduced by a factor of six, while the processing speed increases by a factor between two and five, without loosing (compression) performance. The main ideas behind our investigations are an efficient representation of the estimated and weighted CTW-probabilities and the use of hashing to create the dynamical context tree structure.

Part 1

General Introduction

CHAPTER 1

Proposal (In Dutch)

1. Titel, Opdrachtgever, Uitvoerders, Datum

TITEL: Complexiteitsreductie van het Context-Tree Weighting Algorithme en Performanceverbetering van Lempel-Ziv Technieken
OPDRACHTGEVER: KPN Research
UITVOERDERS: Vakgroep Informatie- en Communicatietheorie, T.U. Eindhoven
DATUM: 17 october, 1995

2. Achtergrond

2.1. Het Context-tree weighting algorithme.

2.1.1. *Eerste resultaten.* Binaire informatie bronnen met een tree-model (zie Rissanen [**27**] en Weinberger, Rissanen en Feder[**48**]) worden gekarakteriseerd door een propere en complete binaire boom, waarbij aan elk blad een parameter gekoppeld is. Deze modelboom bepaalt de kansverdeling volgens welke de bron het volgende symbool (x_t), genereert. Dit geschiedt op de volgende manier. We starten in de wortel van de boom. Het rijtje $x_{t-1}, x_{t-2}, x_{t-3}, \cdots$ beschrijft, omdat de boom proper en compleet is, eenduidig een pad in de boom dat naar een blad leidt. De parameter gekoppeld aan dit blad, geeft aan met welke kans de bron nu een 1 genereert. Voor bronnen met een tree-model geldt dus dat een nieuw symbool statistisch gezien afhankelijk is van een aantal, meest recente, symbolen (context symbolen) uit het verleden.

Datacompressie-algorithmes voor bronnen met een (onbekend) tree-model, maakten tot voor kort (zie Rissanen [**25**] en Weinberger, Rissanen en Feder[**46**]) voor elk nieuw symbool een schatting van de relevante context-lengte, d.w.z. ze probeerden deze lengte zo te kiezen dat de context naar een blad van het werkelijke tree-model zou leiden. Uitgaande van deze lengte werd dan de kans op een 1 voor het nieuwe symbool geschat en daarna gebruikt om (arithmetisch) te coderen. Er kon bewezen worden dat, wanneer de bronrijlengte maar bleef groeien, men na verloop van tijd altijd in staat moest zijn om de relevante context-lengte correct te schatten, waarmee men asymptotisch optimaal redundantiegedrag verkreeg, ongeacht het werkelijke model.

Aan het context-tree weighting (CTW-)algorithme, ontwikkeld in de vakgroep EI [**56**], [**57**],[**53**] ligt een ander principe ten grondslag. Dit algorithme tracht niet telkens de relevante context-lengte te schatten, maar middelt op een slimme manier over alle mogelijke contexten. Dit middelen gebeurt recursief in een context-boom, door gebruik te maken van een wel zeer eenvoudige operatie (vermenigvuldigen, optellen en delen door 2). Per verwerkt bronsymbool zijn slechts enkele operaties van dit type nodig. Een belangrijke eigenschap van het CTW-algorithme is dat deze methode op een natuurlijke en zeer eenvoudige manier geanalyseerd kan worden. Uit deze analyse blijkt dat het algorithme niet alleen voor asymptotisch lange bronrijen(data) goed presteert maar ook voor eindige bronrijtjes de gewenste performance (compactie) geeft.

2.1.2. *Verdere ontwikkelingen.* Na de eerste resultaten voor het CTW-algorithme werd het onderzoek uitgebreid in een aantal richtingen. Belangrijk is de generalisatie van het CTW-algorithme naar algemenere context algorithmen (zie Willems, Shtarkov en Tjalkens[**58**]). Terwijl voor tree-modellen de volgorde van de contextsymbolen langs het pad van wortel van de model-boom naar een blad vastliggen, is er ook een model-klasse denkbaar waarbij deze volgorde nog kan variëren. In elke knoop van de model-boom bepaalt nu een contextsymbool met een bepaald label, hoe we verder gaan, richting bladeren. Ook voor deze bronnen kan een recursief-werkend context-weighting algorithme worden geformuleerd. De analyse is ook nu weer eenvoudig, de performance zoals gewenst. Dit is slechts een eerste generalisatie van de tree model-klasse; er zijn overigens nog twee verdere generalisaties beschreven in [**58**].

Het toepassen van weighting-technieken bleek ook voordelen op te leveren bij het coderen van geheugenloze bronnen die niet stationair zijn. Zo werden voor deze bronnen weighting-algorithmen geformuleerd in Willems[**55**] die optimaal bleken te zijn omdat ze de lower bound van Merhav[**20**] bereikten.

Het context-tree weighting algorithme geniet ook veel belangstelling van andere onderzoeksinstellingen vooral in Japan maar ook in de U.S.A. (zie [**9**], [**13**], [**17**], [**18**], [**19**], [**39**], [**40**], [**41**]).

2.2. Lempel-Ziv algorithmen. Lempel-Ziv algorithmen zijn adaptieve compactie schemas waarin data gecomprimeerd wordt door te verwijzen naar reeds eerder opgetreden subrijen in de data. Deze subrijen vormen een dictionaire van rijen die mogelijk geparsed kunnen worden. Het fundamentele werk van Ziv en Lempel resulteerde in twee basis schemas, nl. het buffer schema [**67**] en het lijst schema [**68**]. Ziv en Lempel bewezen voor het lijst schema dat dit de optimale compactie, d.i. de bron entropie, bereikt voor elke stationaire ergodische bron in de limiet als de datarij lengte oneindig groot wordt. Voor het buffer algorithme duurde het wat langer om de optimaliteit aan te tonen, maar door gebruik te maken van repetitietijden (zie Willems [**50**] en Wyner and Ziv [**61**]) waren Wyner en Ziv in 1994 in staat om dit probleem op te lossen [**63**],[**64**].

Het feit dat de compactie van actuele datarijen redelijk is en de belofte van een asymptotisch optimale compactie, samen met de lage complexiteit van deze algorithmen hebben ertoe geleid dat van Lempel-Ziv afgeleide algorithmen momenteel veel toegepast worden, b.v. in V.42 bis en compactieprogrammas als UNIX COMPRESS en Stacker.

3. Vorig Project

In een vorig project heeft de vakgroep Informatie- en Communicatietheorie voor KPN Research een onderzoek gedaan naar de prestaties van het nieuwe CTW-algorithme en de bekende Lempel-Ziv methodes. Daarbij waren de criteria performance (compactieverhouding) en complexiteit (geheugengebruik, rekensnelheid). Belangrijkste conclusie (zie [**35**]) uit het vorige onderzoek was dat bij eenzelfde (niet al te grote) hoeveelheid geheugen, Lempel-Ziv methodes en het CTW-algorithme bij benadering dezelfde compactieresultaten geven. Krijgt men echter de beschikknig over meer geheugen dan blijken Lempel-Ziv technieken niet tot betere compactie in staat, terwijl het CTW-algorithme wel een betere compactie laat zien.

De experimenten werden uitgevoerd op het Calgary corpus; dit is een verzameling ASCII files. Lempel-Ziv methodes zijn uitermate geschikt voor compactie van ASCII files. Het in principe binaire CTW algorithme kan gemakkelijk gegeneraliseerd worden naar alfabetgrootte 128, maar dit levert niet de gewenste compactie-resultaten op. Het bleek beter te zijn de ASCII symbolen te beschouwen als rijtjes van 7 binaire symbolen, die dan elk met een eigen CTW algorithme worden behandeld. De context (het verleden) voor elk binair symbool bestaat nu uit alle vorige binaire symbolen. Voor het schatten moesten we gebruik maken van een aangepaste Krichevski-Trofimov schatter. Deze heeft de eigenschap dat hij een constante redundantie geeft wanneer er een all-zero of een all-one rij wordt gezien, wat resulteert in een betere compactie bij deterministisch gedrag. Verder was het zinvol alleen maar te wegen op ASCII-grenzen. Dit leidt tot een lagere model-redundantie.

4. Voorstel Nieuw Project

4.1. Complexiteitsreductie van het CTW-algorithme. Omdat de geheugencomplexiteit van het CTW-algorithme groot is, is er gezocht naar manieren om deze complexiteit te verlagen. Uit voorstudie bleek dat er in principe twee mogelijkheden zijn:

- Het weglaten van stukken van de context-boom. Het blijkt dat bepaalde subbomen irrelevant zijn omdat ze corresponderen met een all-zero of all-one subsequence. Het aantal overbodige knopen is vaak meer dan 2/3 van het totale aantal. De compactieverhouding wordt wel een beetje lager door het weglaten van "overbodige" knopen.
- Het efficienter representeren van weeg- en geschatte kansen. Het blijkt dat in principe de verhouding van weeg- en geschatte kans volstaat om het CTW-algorithme uit te kunnen voeren. Deze verhouding kan daarbij het beste logarithmisch gerepresenteerd worden. Verder loont het de moeite om te onderzoeken of de zero- en one-counts niet gezamelijk op een efficientere manier kunnen worden opgeslagen. Daarnaast blijkt een lijst-implementatie i.p.v. een binaire boom implementatie in elke ASCII-knoop een geheugenbesparing op te leveren.

  Deze observaties geven een geheugencomplexiteitsreductie van ongeveer een factor 2, misschien zelfs 3. De aanpassingen hoeven de compactieverhouding niet nadelig te beinvloeden.

Concluderend zou een totale geheugencomplexiteitsreductie van een factor 5 tot de mogelijkheden moeten behoren, bij gelijkblijvende compactie. Daarnaast is het zo dat de logarithmische representatie een snelheidsverbetering tot gevolg zal hebben. Dit alles zal geanalyseerd, geprogrammeerd en gesimuleerd worden in het kader van dit project.

4.2. Performanceverbetering bij Lempel-Ziv technieken. Tot voor kort was er een duidelijke verschil in performance en complexiteit tussen Lempel-Ziv algorithmes ([**67**],[**68**]) en context methoden ([**25**],[**46**],[**48**],[**57**]). De performance (compactie) van Lempel-Ziv algorithmes is dan wel slechter, maar de complexiteit (verwerkingssnelheid, geheugenbeslag) is ook beter dan bij context algorithmes. Omdat de prijs van halfgeleidergeheugens echter steeds daalt en processoren steeds sneller worden, zullen context algorithmes in de toekomst waarschijnlijk de overhand gaan krijgen en Lempel-Ziv algorithmes gaan verdwijnen.

Recentelijk op het 1995 IEEE International Symposium on Information Theory heeft Ziv echter laten zien dat het mogelijk is een combinatie te maken van Lempel-Ziv en context algorithmes[**66**],[**10**]. De overigens nog niet uitgewerkte ideeen van Ziv leiden mogelijk tot een nieuwe generatie van algorithmes die aan de ene kant de lagere complexiteit van Lempel-Ziv algorithmes hebben, maar die daarnaast ook de goede performance van context algorithmes bezitten.

In het kader van dit vervolgproject zal onderzocht worden wat deze synthese kan opleveren. Aan de ene kant zullen Lempel-Ziv algorithmes (nogmaals) bestudeerd moeten worden, waarbij in de analyse de conditionele versie van Kac's stelling een voorname rol zal spelen. Aan de andere kant zullen context-ideeen op een juiste manier ingepast moeten worden. Het is vanzelfsprekend dat daarbij performance en complexiteit tegen elkaar afgewogen moeten worden. Literatuurverkenning is natuurlijk een onderdeel van dit onderzoek dat uiteindelijk moet leiden tot uitspraken over deze nieuwe hybride algorithmen.

CHAPTER 2

Executive Summary CTW-1 Project

In this report we describe the results of a series of experiments that were done to compare the performance of Lempel-Ziv based data compression algorithms and the recently developed context-tree weighting method.

The Lempel-Ziv algorithms encode a next part of the data by replacing it by a code that refers to an earlier occurence of this part. These methods can be divided in two classes. First there are the buffer methods in which a buffer is used to look for earlier occurences. These algorithms are based on [**67**]. The methods in the second class, the tree algorithms, are all based on [**68**]. These methods grow a tree of strings that have occurred earlier in the data. The V42-bis standard is also Lempel-Ziv based. Techniques from both the buffer and the tree method are combined in this implementation.

The context-tree weighting method was developed in the Information and Communication Theory Group at Eindhoven University of Technology [**56**]. It is a statistical data compression algorithm, i.e. it tries to approximate the statistic of the data by counting occurrences of symbols, given the preceeding symbols (their context). Using this statistic an arithmethic code can do the actual encoding and decoding. A problem with these methods always was to determine for each symbol how large the corresponding context length should be. By using a recursive weighting technique, the context-tree weighting method avoids this problem and finds a good statistic.

Before starting the experiments it was well-known that Lempel-Ziv methods combine an acceptable compression-rate with a low storage complexity and a high processing speed. From preliminary experiments it became clear that the context-tree weighting method had the advantage of having an excellent compression-rate but also the disadvantages of requiring more memory and having a lower processing speed. It was the objective of the present project to compare the compression-rate versus memory trade-offs of both the Lempel-Ziv methods and the context-tree weighting method. The files in the Calgary Corpus were used as test-data.

Five Lempel-Ziv variants were implemented in software, the buffer methods LZ77 and LZSS, and the tree methods LZ78, LZW, and LZWE. The experiments showed that they achieve compression-rates not smaller than 3 bit per symbol for the text files in the Calgary Corpus. Processing speed is between 20 K (for encoding for buffer methods) and 400 K symbols per second (for tree methods). The required memory is usually between 10 kbyte and 1Mbyte.

The contex-tree weighting method was implemented using binary decomposition, weighting only at byte boundaries, zero-redundancy estimation, binary search trees, pruning of unique contex paths, and the missing-context idea. This implementation can achieve compression-rates below 2 bit per symbol for large text files in the Calgary Corpus. The encoding and decoding speed is between 1 K and 2 K symbols per second. The amount of memory used is typically between 1 Mbyte and 100 Mbyte.

Increasing the amount of memory for Lempel-Ziv algorithm above 1 Mbyte does not give a better compression-rate. Decreasing the available memory for the context-tree weighting method below 1 Mbyte makes the compression-rate worse (as expected). If for both classes of algorithms the amount of memory is the same and in the range from 10 kbyte to 1 Mbyte, the compression-rates for both classes are comparable. This is without doubt the most important conclusion of the project. Taking into account that the cost of storage is decreasing and that larger amounts of memory are typical in the future, one will stop to use Lempel-Ziv algorithms like V42-bis since their compression-rate will not decresase any further and start to use statistical methods like the contex-tree weighting algorithm instead. The contex-tree weighting method achieves the desired decrease in compression-rate.

Although these first experiments for the context-tree weighting method show the superiority of this algorithm, it should be recognized that the development of the context-tree weighting algorithm has just started. In the future the objective must be to develop new methods for and implementations of the context-tree weighting method that decrease the amount of memory needed and that increase the speed of the algorithm.

CHAPTER 3

The Context Tree Weighting Method: Basic Properties

AUTHORS: Frans M.J. Willems, Yuri M. Shtarkov[1] and Tjalling J. Tjalkens

(This chapter is almost identical to [57].)

Abstract

We describe a sequential universal data compression procedure for binary tree sources that performs the "double mixture". Using a context tree, this method weights in an efficient recursive way the coding distributions corresponding to all bounded memory tree sources, and achieves a desirable coding distribution for tree sources with an unknown model and unknown parameters. Computational and storage complexity of the proposed procedure are both linear in the source sequence length. We derive a natural upper bound on the cumulative redundancy of our method for individual sequences. The three terms in this bound can be identified as coding, parameter and model redundancy. The bound holds for all source sequence lengths, not only for asymptotically large lengths. The analysis that leads to this bound is based on standard techniques and turns out to be extremely simple. Our upper bound on the redundancy shows that the proposed context tree weighting procedure is optimal in the sense that it achieves the Rissanen (1984) lower bound.

Keywords

Sequential data compression, universal source coding, tree sources, modeling procedure, arithmetic coding, cumulative redundancy bounds.

1. Introduction, Concepts

A *finite memory tree source* has the property that the next-symbol probabilities depend on a finite number of most recent symbols. This number in general depends on the actual values of these most recent symbols. Binary sequential universal source coding procedures for finite memory tree sources often make use of a context tree which contains for each string (context) the number of zeros and the number of ones that have followed this context, in the source sequence seen so far. The standard approach (see e.g. Rissanen and Langdon[**29**], Rissanen[**25**],[**27**], and Weinberger, Lempel and Ziv[**46**]) is that, given the past source symbols, one uses this context tree to estimate the actual 'state' of the finite memory tree source. Subsequently this state is used to estimate the distribution that generates the next source symbol. This estimated distribution can be used in arithmetic coding procedures (see e.g. Rissanen and Langdon[**29**]) to encode (and decode) the next source symbol efficiently, i.e. with low complexity and with negligible additional redundancy.

[1]Institute for Problems of Information Transmission, Ermolovoystr. 19, 101447, Moscow, GSP-4.

After Rissanen's pioneering work in [**25**], Weinberger, Lempel and Ziv[**46**] developed a procedure that achieves optimal exponential decay of the error probability in estimating the current state of the tree source. These authors were also able to demonstrate that their coding procedure achieves asymptotically the lower bound on the average redundancy, as stated by Rissanen ([**26**], theorem 1, or [**27**], theorem 1). Recently Weinberger, Rissanen and Feder[**48**] could prove the optimality, in the sense of achieving Rissanen's lower bound on the redundancy, of an algorithm similar to that of Rissanen in [**25**].

An unpleasant fact about the standard approach is that one has to specify parameters (α and β in Rissanen's procedure [**25**] or K for the Weinberger, Lempel and Ziv[**46**] method), that do not affect the asymptotical performance of the procedure, but may have a big influence on the behavior for finite (and realistic) source sequence lengths. These artificial parameters are necessary to regulate the state estimation characteristics. This gave the authors the idea that the state estimation concept may not be as natural as one believes. A better starting principle would be, just to find a good coding distribution. This more or less trivial guideline immediately suggests the application of *model weighting techniques*. An advantage of weighting procedures is that they perform well not only on the average but for each individual sequence. Model weighting (twice-universal coding) is not new. It was first suggested by Ryabko[**31**] for the class of finite order Markov sources (see also [**32**] for a similar approach to prediction). The known literature on model weighting resulted however in probability assignments that require complicated sequential updating procedures. Instead of finding implementable coding methods one concentrated on achieving low redundancies. In what follows we will describe a probability assignment for *bounded memory tree sources* that allows efficient updating. This procedure, which is based on tree-recursive model-weighting, results in a coding method that is very easy to analyze, and that has a desirable performance, both in realized redundancy and in complexity.

2. Binary Bounded Memory Tree Sources

2.1. Strings. A string s is a concatenation of binary symbols, hence $s = q_{1-l}q_{2-l}\cdots q_0$ with $q_{-i} \in \{0,1\}$ for $i = 0, 1, \cdots, l-1$. Note that we index the symbols in the string from right to left, starting with 0 and going negative. For the length of a string s we write $l(s)$. A semi-infinite string $s = \cdots q_{-1}q_0$ has length $l(s) = \infty$. The empty string λ has length $l(\lambda) = 0$.

If we have two strings $s' = q'_{1-l'}q'_{2-l'}\cdots q'_0$ and $s = q_{1-l}q_{2-l}\cdots q_0$ then $s's \triangleq q'_{1-l'}q'_{2-l'}\cdots q'_0 q_{1-l}q_{2-l}\cdots q_0$ is the concatenation of both. If $\mathcal{V}$ is a set of strings and $q \in \{0,1\}$, then $\mathcal{V} \times q \triangleq \{vq : v \in \mathcal{V}\}$.

We say that a string $s = q_{1-l}q_{2-l}\cdots q_0$ is a *suffix* of the string $s' = q'_{1-l'}q'_{2-l'}\cdots q'_0$ if $l \leq l'$ and $q_{-i} = q'_{-i}$ for $i = 0, l-1$. The empty string λ is a suffix of all strings.

2.2. Binary bounded memory tree source definition. A *binary tree source* generates a sequence $x_{-\infty}^{\infty}$ of digits assuming values in the alphabet $\{0,1\}$. We denote by x_m^n the sequence $x_m x_{m+1}\cdots x_n$, and allow m and n to be infinitely large. For $n < m$ the sequence x_m^n is empty, denoted by ϕ.

The statistical behavior of a binary *finite memory* tree source can be described by means of a *suffix set* $\mathcal{S}$. This suffix set is a collection of binary strings $s(k)$, with $k = 1, 2, \cdots, |\mathcal{S}|$.

We require it to be *proper* and *complete*. Properness of the suffix set implies that no string in $\mathcal{S}$ is a *suffix* of any other string in $\mathcal{S}$. Completeness guarantees that each semi-infinite sequence (string) $\cdots x_{n-2}x_{n-1}x_n$ has a suffix that belongs to $\mathcal{S}$. This suffix is unique since $\mathcal{S}$ is proper.

Let $D \in \{0, 1, \cdots\}$ be fixed throughout this paper. A *bounded memory* tree source has a suffix set $\mathcal{S}$ that satisfies $l(s) \leq D$ for all $s \in \mathcal{S}$. We say that the source has memory not larger than D.

The properness and completeness of the suffix set make it possible to define the *suffix function* $\beta_{\mathcal{S}}(\cdot)$. This function maps semi-infinite sequences onto their unique suffix s in $\mathcal{S}$. Since all suffixes in $\mathcal{S}$ have length not larger than D, only the last D symbols of a semi-infinite sequence determine its suffix in $\mathcal{S}$. To each suffix s in $\mathcal{S}$ there corresponds a *parameter* θ_s. Each parameter (i.e. the probability of a source symbol being one) assumes a value in $[0, 1]$ and specifies a probability distribution over $\{0, 1\}$. Together, all parameters form the parameter vector $\Theta_{\mathcal{S}} \triangleq \{\theta_s : s \in \mathcal{S}\}$. If the tree source has emitted the semi-infinite sequence $x_{-\infty}^{t-1}$ up to now, the suffix function tells us that the parameter for generating the next binary digit x_t of the source is θ_s, where $s = \beta_{\mathcal{S}}(x_{t-D}^{t-1})$. Thus

DEFINITION 2.1. The *actual* next-symbol probabilities for a bounded memory tree source with suffix set $\mathcal{S}$ and parameter vector $\Theta_{\mathcal{S}}$ are

$$P_a(X_t = 1|x_{t-D}^{t-1}, \mathcal{S}, \Theta_{\mathcal{S}}) = 1 - P_a(X_t = 0|x_{t-D}^{t-1}, \mathcal{S}, \Theta_{\mathcal{S}}) \triangleq \theta_{\beta_{\mathcal{S}}(x_{t-D}^{t-1})}, \text{ for all } t. \tag{1}$$

The actual block probabilities are now products of actual next-symbol probabilities, i.e. $P_a(X_1^t = x_1^t|x_{1-D}^0, \mathcal{S}, \Theta_{\mathcal{S}}) = \Pi_{\tau=1}^t P_a(X_\tau = x_\tau|x_{\tau-D}^{\tau-1}, \mathcal{S}, \Theta_{\mathcal{S}})$.

All tree sources with the same suffix set are said to have the same *model.* Model and suffix set are equivalent. The set of all tree models having memory not larger than D is called the *model class* $\mathcal{C}_D$. It is possible to specify a model in this model class by a *natural* code by encoding the suffix set $\mathcal{S}$ recursively. The code of $\mathcal{S}$ is the code of the empty string λ. The code of a string s is void if $l(s) = D$, otherwise it is 0 if $s \in \mathcal{S}$ and 1 followed by the codes of the strings $0s$ and $1s$ if $s \notin \mathcal{S}$. If we use this natural code, the number of bits that are needed to specify a model $\mathcal{S} \in \mathcal{C}_D$ is equal to $\Gamma_D(\mathcal{S})$, where

DEFINITION 2.2. $\Gamma_D(\mathcal{S})$, the cost of a model $\mathcal{S}$ with respect to model class $\mathcal{C}_D$, is defined as

$$\Gamma_D(\mathcal{S}) \triangleq |\mathcal{S}| - 1 + |\{s : s \in \mathcal{S}, l(s) \neq D\}|, \tag{2}$$

where it is assumed that $\mathcal{S} \in \mathcal{C}_D$.

EXAMPLE 2.1. Let $D = 3$. Consider a source with suffix set $\mathcal{S} = \{00, 10, 1\}$ and parameters $\theta_{00} = 0.5$, $\theta_{10} = 0.3$, and $\theta_1 = 0.1$ (see Figure 3.1). The (conditional) probability of the source generating the sequence 0110100 given the past symbols $\cdots 010$ can be calculated as follows :

$$P_a(0110100|\cdots 010) = (1 - \theta_{10})\theta_{00}\theta_1(1 - \theta_1)\theta_{10}(1 - \theta_1)(1 - \theta_{10}) = 0.0059535. \tag{3}$$

Since $D = 3$, the model (suffix set) $\mathcal{S}$ can be specified by

$$\text{code}(\mathcal{S}) = \text{code}(\lambda) = 1 \ \text{code}(0) \ \text{code}(1) = 1 \ 1 \ \text{code}(00) \ \text{code}(10) \ 0 = 1 \ 1 \ 0 \ 0 \ 0. \tag{4}$$

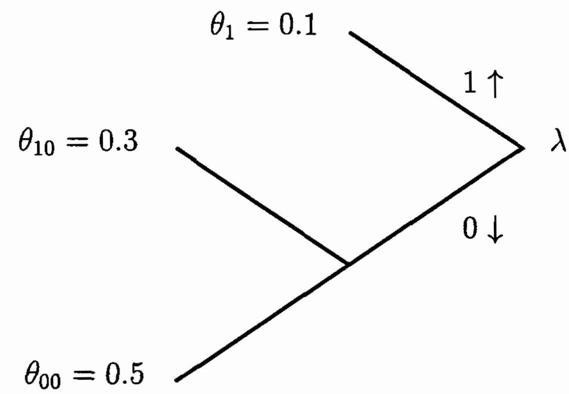


FIGURE 3.1. Model (suffix set) and parameters.

Tree sources are related to FSMX sources that were first described by Rissanen[**27**]. FSMX sources can be considered as tree sources whose suffix set $\mathcal{S}$ is *closed.* A suffix set is said to be closed if the *generator* of each suffix $s \in \mathcal{S}$ belongs to $\mathcal{S}$ or is a suffix of an $s \in \mathcal{S}$. The generator of a suffix $s = q_{1-l} \cdots q_{-1} q_0$ is $q_{1-l} \cdots q_{-1}$. Note that $\mathcal{S} = \{00, 010, 110, 1\}$ is a tree model, but not an FSMX model. Each finite memory tree source with suffix set $\mathcal{S}$ has a finite state machine implementation. The number of states is then $|\mathcal{S}|$ or more. Only for tree sources with a closed suffix set $\mathcal{S}$ (i.e. for FSMX sources) the number of states is equal to $|\mathcal{S}|$.

3. Codes and Redundancy

Let $T \in \{1, 2, \cdots\}$. Instead of the source sequence $x_1^T \triangleq x_1 x_2 \cdots x_T$ itself, the *encoder* sends a codeword $c^L \triangleq c_1 c_2 \cdots c_L$ consisting of $\{0,1\}$-components to the *decoder.* The decoder must be able to reconstruct the source sequence x_1^T from this codeword.

We assume that both the encoder and the decoder have access to the past source symbols $x_{1-D}^0 = x_{1-D} \cdots x_{-1} x_0$, so that implicitely the suffix that determines the probability distribution of the first source symbols, is available to them. A codeword that is formed by the encoder therefore depends not only on the source sequence x_1^T but also on x_{1-D}^0. To denote this functional relationship we write $c^L(x_1^T | x_{1-D}^0)$. The length of the codeword, in binary digits, is denoted as $L(x_1^T | x_{1-D}^0)$.

We restrict ourselves to *prefix codes* here (see [4], chapter 5). These codes are not only uniquely decodable but also *instantaneous* or *self-punctuating* which implies that you can immediately recognize a codeword when you see it. The set of codewords that can be produced for a fixed x_{1-D}^0 form a prefix code, i.e. no codeword is the prefix of any other codeword in this set. All sequences $c_1 c_2 \cdots c_l$, for some $l = 1, 2, \cdots, L$ are a prefix of $c_1 c_2 \cdots c_L$.

The codeword lengths $L(x_1^T | x_{1-D}^0)$ determine the *individual redundancies.*

DEFINITION 3.1. The individual redundancy $\rho(x_1^T | x_{1-D}^0, \mathcal{S}, \Theta_{\mathcal{S}})$ of a sequence x_1^T given the past symbols x_{1-D}^0, with respect to a source with model $\mathcal{S} \in \mathcal{C}_D$ and parameter vector

Θ_S, is defined as[2]

$$\rho(x_1^T|x_{1-D}^0, \mathcal{S}, \Theta_S) \triangleq L(x_1^T|x_{1-D}^0) - \log \frac{1}{P_a(x_1^T|x_{1-D}^0, \mathcal{S}, \Theta_S)}, \tag{5}$$

where $L(x_1^T|x_{1-D}^0)$ is the length of the codeword that corresponds to x_1^T given x_{1-D}^0. We consider only sequences x_1^T with positive probability $P_a(x_1^T|x_{1-D}^0, \mathcal{S}, \Theta_S)$.

The value $\log(1/P_a(x_1^T|x_{1-D}^0, \mathcal{S}, \Theta_S))$ can be regarded as the *information* contained in x_1^T given the past x_{1-D}^0. It is often called the *ideal codeword length.* Note that we do not divide the redundancies by the source sequence length T, we consider only *cumulative* redundancies. Note also that our redundancies can be negative.

The objective in universal source coding is to design methods that achieve small individual redundancies with respect to all sources in a given class. Since it is also very important that these methods have low (storage and computational) complexity, it would be more appropriate to say that the emphasis in source coding is on finding a desirable trade-off between achieving small redundancies and keeping the complexity low.

4. Arithmetic Coding

An *arithmetic* encoder *computes* the codeword that corresponds to the actual source sequence. The corresponding decoder reconstructs the actual source sequence from this codeword again by computation. Using arithmetic codes it is possible to process source sequences with a large length T. This is often needed to reduce the redundancy per source symbol.

Arithmetic codes are based on the Elias algorithm (unpublished, but described by Abramson[**1**] and Jelinek[**12**]) or on enumeration (e.g. Schalkwijk[**34**] and Cover[**3**]). Arithmetic coding became feasable only after Rissanen[**24**], and Pasco[**21**], had solved the accuracy issues that were involved. We will not discuss such issues here. Instead we will assume that all computations are carried out with *infinite precision.*

Suppose that the encoder and decoder both have access to, what is called the *coding distribution* $P_c(x_1^t), x_1^t \in \{0,1\}^t, t = 0, 1, \cdots, T$. We require that this distibution satisfies

$$\begin{aligned} P_c(\phi) &= 1, \\ P_c(x_1^{t-1}) &= P_c(x_1^{t-1}, X_t = 0) + P_c(x_1^{t-1}, X_t = 1), \text{ for all } x_1^{t-1} \in \{0,1\}^{t-1}, t = 1, \cdots, T, \\ \text{and } P_c(x_1^T) &> 0 \text{ for all } \textit{possible } x_1^T \in \{0,1\}^T, \end{aligned} \tag{6}$$

where possible sequences are sequences that can actually occur, i.e. sequences x_1^T with $P_a(x_1^T) > 0$. Note that ϕ stands for the empty sequence (x_1^0).

In appendix 9 we describe the Elias algorithm. It results in the following theorem.

THEOREM 4.1. *Given a coding distribution $P_c(x_1^t), x_1^t \in \{0,1\}^t, t = 0, 1, \cdots, T$, the Elias algorithm achieves codeword lengths $L(x_1^T)$ that satisfy*

$$L(x_1^T) < \log \frac{1}{P_c(x_1^T)} + 2, \tag{7}$$

for all possible $x_1^T \in \{0,1\}^T$. The codewords form a prefix code.

[2]The basis of the $\log(\cdot)$ is assumed to be 2, throughout this paper.

The difference between the codeword length $L(x_1^T)$ and $\log(1/P_c(x_1^T))$ is always less than 2 bits. We say that the individual *coding redundancy* is less than 2 bits.

We conclude this section with the observation that the Elias algorithm combines an acceptable coding redundancy with a desirable *sequential* implementation. The number of operations is linear in the source sequence length T. It is crucial however that the encoder and decoder have access to the probabilities $P_c(x_1^{t-1}, X_t = 0)$ and $P_c(x_1^{t-1}, X_t = 1)$ after having processed $x_1x_2 \cdots x_{t-1}$. If this is the case we say that the coding distribution is *sequentially available.*

It should be noted that our view of an arithmetic code is slightly different from usual. We assume that block probabilities are fed into the encoder and decoder and not conditional probabilities as usual. The reason for this is that it creates a better match between our modeling algorithm and the arithmetic code, and avoids multiplications.

If we are ready to accept a loss of at most 2 bits coding redundancy, we are now left with the problem of finding good, sequentially available, coding distributions.

5. Probability Estimation

The probability that a memoryless source with parameter θ generates a sequence with a zeros and b ones is $(1-\theta)^a\theta^b$. If we *weight* this probability over all θ with a $(\frac{1}{2}, \frac{1}{2})$-Dirichlet distribution we obtain the so-called Krichevsky-Trofimov estimate (see [**15**]).

DEFINITION 5.1. The Krichevski-Trofimov(KT) estimated probability for a sequence containing $a \geq 0$ zeros and $b \geq 0$ ones is defined as

$$P_e(a,b) \triangleq \int_0^1 \frac{1}{\pi\sqrt{(1-\theta)\theta}}(1-\theta)^a\theta^b d\theta. \tag{8}$$

This estimator has properties that are listed in the lemma that follows. The lemma is proved in appendix 10.

LEMMA 5.1. *The KT-probability estimator $P_e(a,b)$*
1. can be computed sequentially, i.e. $P_e(0,0) = 1$, and for $a \geq 0$ and $b \geq 0$

$$P_e(a+1,b) = \frac{a+\frac{1}{2}}{a+b+1} \cdot P_e(a,b) \text{ and } P_e(a,b+1) = \frac{b+\frac{1}{2}}{a+b+1} \cdot P_e(a,b), \tag{9}$$

2. satisfies, for $a+b \geq 1$, the following inequality

$$P_e(a,b) \geq \frac{1}{2} \cdot \frac{1}{\sqrt{a+b}}(\frac{a}{a+b})^a(\frac{b}{a+b})^b. \tag{10}$$

The sequential behavior of the KT-estimator was studied by Shtarkov[**37**]. An other estimator, the Laplace estimator, is investigated by Rissanen[**27**],[**28**]. This estimator can be obtained by weighting $(1-\theta)^a\theta^b$ with θ uniform over $[0,1]$.

For the KT-estimator the *parameter redundancy* can be uniformly bounded, using the lowerbound (see lemma 5.1) on $P_e(a,b)$, i.e.

$$\log\frac{(1-\theta)^a\theta^b}{P_e(a,b)} \leq \log\frac{(1-\theta)^a\theta^b}{\frac{1}{2}\frac{1}{\sqrt{a+b}}(\frac{a}{a+b})^a(\frac{b}{a+b})^b} \leq \frac{1}{2}\log(a+b)+1, \tag{11}$$

for all $a+b \geq 1$ and all $\theta \in [0,1]$. It is impossible to prove such a uniform bound for the Laplace estimator.

6. Coding for an Unknown Tree Source

6.1. Definition of the context tree weighting method. Consider the case where we have to compress a sequence which is (supposed to be) generated by a tree source, whose suffix set $\mathcal{S} \in \mathcal{C}_D$ and parameter vector $\Theta_{\mathcal{S}}$ are unknown to the encoder and the decoder. We will define a *weighted* coding distribution for this situation, study its performance and discuss its implementation. The coding distribution is based on the concept of a *context tree* (see Figure 3.2).

DEFINITION 6.1. The context tree $\mathcal{T}_D$ is a set of *nodes* labeled s, where s is a (binary) string with length $l(s)$ such that $0 \leq l(s) \leq D$. Each node $s \in \mathcal{T}_D$ with $l(s) < D$, 'splits up' in two nodes, $0s$ and $1s$. The node s is called the *parent* of the nodes $0s$ and $1s$, who in turn are the *children* of s. To each node $s \in \mathcal{T}_D$, there correspond counts $a_s \geq 0$ and $b_s \geq 0$. For the children $0s$ and $1s$ of parent node s, the counts must satisfy $a_{0s} + a_{1s} = a_s$ and $b_{0s} + b_{1s} = b_s$.

Now, to each node there corresponds a *weighted probability.* This weighted probability is defined recursively on the context tree $\mathcal{T}_D$. Without any doubt, this is the basic definition in this paper.

DEFINITION 6.2. To each node $s \in \mathcal{T}_D$, we assign a weighted probability P_w^s which is defined as

$$P_w^s \triangleq \begin{cases} \frac{1}{2}P_e(a_s, b_s) + \frac{1}{2}P_w^{0s}P_w^{1s} & \text{for } 0 \leq l(s) < D, \\ P_e(a_s, b_s) & \text{for } l(s) = D. \end{cases} \tag{12}$$

The context tree together with the weighted probability distributions of the nodes is called a *weighted context tree.*

This definition shows a weighting of both the estimated probability in a node and the product of the weighted probabilities that correspond to its children. The next lemma gives another way of looking at the weighting that is performed in (12). It explains that a weighted probability of a node can be regarded as a weighting over the estimated probabilities corresponding to all (sub-)models that live above this node. The cost (see (2)) of a (sub-)model determines its weighting factor. The proof of this lemma can be found in appendix 11.

LEMMA 6.1. *The weighted probability P_w^s of a node $s \in \mathcal{T}_D$ with $l(s) = d$ for $0 \leq d \leq D$ satisfies*

$$P_w^s = \sum_{\mathcal{U} \in \mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e(a_{us}, b_{us}), \tag{13}$$

with $\sum_{\mathcal{U} \in \mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} = 1$. The summation is over all complete and proper suffix sets $\mathcal{U}$.

To be able to define a weighted coding distribution, we assume that the counts (a_s, b_s), $s \in \mathcal{T}_D$ are determined by the source sequence x_1^t seen up to now, assuming that x_{1-D}^0 are the past symbols.

DEFINITION 6.3. For each $s \in \mathcal{T}_D$ let $a_s(x_1^t|x_{1-D}^0)$, respectively $b_s(x_1^t|x_{1-D}^0)$, be the number of times that $x_\tau = 0$, respectively $x_\tau = 1$, in x_1^t for $1 \leq \tau \leq t$ such that $x_{\tau-l(s)}^{\tau-1} = s$. The weighted probabilities corresponding to the nodes $s \in \mathcal{T}_D$ are now denoted by

$P_w^s(x_1^t|x_{1-D}^0)$. For any sequence x_{1-D}^0 of past symbols, we define our weighted coding distribution as

$$P_c(x_1^t|x_{1-D}^0) \triangleq P_w^\lambda(x_1^t|x_{1-D}^0), \tag{14}$$

for all $x_1^t \in \{0,1\}^t, t = 0, 1, \cdots, T$, where λ is the root node of the context tree $\mathcal{T}_D$.

This coding distribution determines the *context tree weighting method.* Note that the counts indeed satisfy the restrictions mentioned in definition 6.1. To verify that it satisfies (6) we formulate a lemma. The proof of this lemma can be found in appendix 12.

LEMMA 6.2. *Let $t = 1, 2, \cdots, T$. If $s \in \mathcal{T}_D$ is not a suffix of x_{t-D}^{t-1}, then*

$$P_w^s(x_1^{t-1}, X_t = 0|x_{1-D}^0) = P_w^s(x_1^{t-1}, X_t = 1|x_{1-D}^0) = P_w^s(x_1^{t-1}|x_{1-D}^0), \tag{15}$$

and, if s is a suffix of x_{t-D}^{t-1}, then

$$P_w^s(x_1^{t-1}, X_t = 0|x_{1-D}^0) + P_w^s(x_1^{t-1}, X_t = 1|x_{1-D}^0) = P_w^s(x_1^{t-1}|x_{1-D}^0). \tag{16}$$

To check that the weighted coding distribution defined in (14) is an allowable coding distribution observe that $P_w^\lambda(\phi|x_{1-D}^0) = 1$. Subsequently, note that lemma 6.2 states that $P_w^\lambda(x_1^{t-1}, X_t = 0|x_{1-D}^0) + P_w^\lambda(x_1^{t-1}, X_t = 1|x_{1-D}^0) = P_w^\lambda(x_1^{t-1}|x_{1-D}^0)$ since λ is a suffix of all strings x_{t-D}^{t-1}. From this we may conclude that our weighted coding distribution satisfies (6) after having verified that weighted probabilities are always positive.

We are now ready to investigate the redundancy of the context tree weighting method.

6.2. An upper bound on the redundancy. First we give a definition.

DEFINITION 6.4. Let

$$\gamma(z) \triangleq \begin{cases} z & \text{for } 0 \leq z < 1 \\ \frac{1}{2}\log z + 1 & \text{for } z \geq 1, \end{cases}$$

hence $\gamma(\cdot)$ is a convex-$\cap$ continuation of $\frac{1}{2}\log z + 1$ for $0 \leq z < 1$ satisfying $\gamma(0) = 0$.

The basic result concerning the context tree weighting technique can be stated now.

THEOREM 6.3. *The individual redundancies with respect to any source with model $\mathcal{S} \in \mathcal{C}_D$ and parameter vector $\Theta_\mathcal{S}$ are upper bounded by*

$$\rho(x_1^T|x_{1-D}^0, \mathcal{S}, \Theta_\mathcal{S}) < \Gamma_D(\mathcal{S}) + |\mathcal{S}|\gamma(\frac{T}{|\mathcal{S}|}) + 2, \tag{17}$$

for all $x_1^T \in \{0,1\}^T$, for any sequence of past symbols x_{1-D}^0, if we use the weighted coding distribution specified in (14).

Note that (17) can be rewritten as

$$\rho(x_1^T|x_{1-D}^0, \mathcal{S}, \Theta_\mathcal{S}) < \begin{cases} \Gamma_D(\mathcal{S}) + T + 2 & \text{for } T = 1, \cdots, |\mathcal{S}| - 1 \\ \Gamma_D(\mathcal{S}) + \frac{|\mathcal{S}|}{2}\log\frac{T}{|\mathcal{S}|} + |\mathcal{S}| + 2 & \text{for } T = |\mathcal{S}|, |\mathcal{S}| + 1, \cdots. \end{cases}$$

The redundancy bound in theorem 6.3 holds with respect to *all* sources with model $\mathcal{S} \in \mathcal{C}_D$ and parameter vector $\Theta_\mathcal{S}$, and *not only the actual source.* Using the definition of redundancy (see (5)) we therefore immediately obtain an upper bound on the codeword lengths.

COROLLARY. *Using the coding distribution in (14), the codeword lengths $L(x_1^T|x_{1-D}^0)$ are upper bounded by*

$$L(x_1^T|x_{1-D}^0) < \min_{\mathcal{S}\in\mathcal{C}_D}\left(\min_{\Theta_\mathcal{S}}\log\frac{1}{P_a(x_1^T|x_{1-D}^0,\mathcal{S},\Theta_\mathcal{S})}+\Gamma_D(\mathcal{S})+|\mathcal{S}|\gamma(\frac{T}{|\mathcal{S}|})\right)+2, \tag{18}$$

for all $x_1^T\in\{0,1\}^T$, for any sequence x_{1-D}^0 of past symbols.

PROOF. Consider a sequence $x_1^T\in\{0,1\}^T$, a suffix set $\mathcal{S}\in\mathcal{C}_D$ and a parameter vector $\Theta_\mathcal{S}$. Let $a_s=a_s(x_1^T|x_{1-D}^0)$ and $b_s=b_s(x_1^T|x_{1-D}^0)$ for all $s\in\mathcal{T}_D$. We split up the individual redundancy in three terms, model redundancy, parameter redundancy and coding redundancy :

$$\begin{aligned}\rho(x_1^T|x_{1-D}^0,\mathcal{S},\Theta_\mathcal{S}) &= L(x_1^T|x_{1-D}^0)-\log\frac{1}{P_a(x_1^T|x_{1-D}^0,\mathcal{S},\Theta_\mathcal{S})}\\ &= \log\frac{\prod_{s\in\mathcal{S}}P_e(a_s,b_s)}{P_c(x_1^T|x_{1-D}^0)}+\log\frac{P_a(x_1^T|x_{1-D}^0,\mathcal{S},\Theta_\mathcal{S})}{\prod_{s\in\mathcal{S}}P_e(a_s,b_s)}\\ &\quad+\left(L(x_1^T|x_{1-D}^0)-\log\frac{1}{P_c(x_1^T|x_{1-D}^0)}\right).\end{aligned} \tag{19}$$

For the last term, the coding redundancy, we obtain, using theorem 4.1, that

$$L(x_1^T|x_{1-D}^0)-\log\frac{1}{P_c(x_1^T|x_{1-D}^0)}<2. \tag{20}$$

We treat the parameter redundancy, the middle term, as follows

$$\begin{aligned}\log\frac{P_a(x_1^T|x_{1-D}^0,\mathcal{S},\Theta_\mathcal{S})}{\prod_{s\in\mathcal{S}}P_e(a_s,b_s)} &= \sum_{s\in\mathcal{S}}\log\frac{(1-\theta_s)^{a_s}\theta_s^{b_s}}{P_e(a_s,b_s)}\\ &\le \sum_{s\in\mathcal{S}:a_s+b_s>0}\left(\frac{1}{2}\log(a_s+b_s)+1\right)\\ &= |\mathcal{S}|\sum_{s\in\mathcal{S}}\frac{1}{|\mathcal{S}|}\gamma(a_s+b_s)\\ &\le |\mathcal{S}|\gamma(\sum_{s\in\mathcal{S}}\frac{a_s+b_s}{|\mathcal{S}|})=|\mathcal{S}|\gamma(\frac{T}{|\mathcal{S}|}).\end{aligned} \tag{21}$$

The product $\prod_{s\in\mathcal{S}}P_e(a_s,b_s)$ makes it possible, to split up the parameter redundancy in $|\mathcal{S}|$ terms representing the parameter redundancies corresponding to each of the $|\mathcal{S}|$ suffixes in $\mathcal{S}$. The term corresponding to suffix $s\in\mathcal{S}$ can be upper bounded by $\frac{1}{2}\log(a_s+b_s)+1$ as we have seen before in (11), however only for $a_s+b_s>0$. For $a_s+b_s=0$ such a term does not contribute to the redundancy. This is why we have introduced the function γ. Its $\cap$-convexity makes it possible to apply Jensen's inequality (see Cover and Thomas[4], p. 25).

What remains to be investigated is the first term in (19), the *model redundancy* term. It follows from lemma 6.1 that

$$P_w^\lambda=\sum_{\mathcal{U}\in\mathcal{C}_D}2^{-\Gamma_D(\mathcal{U})}\prod_{s\in\mathcal{U}}P_e(a_s,b_s)\ge 2^{-\Gamma_D(\mathcal{S})}\prod_{s\in\mathcal{S}}P_e(a_s,b_s). \tag{22}$$

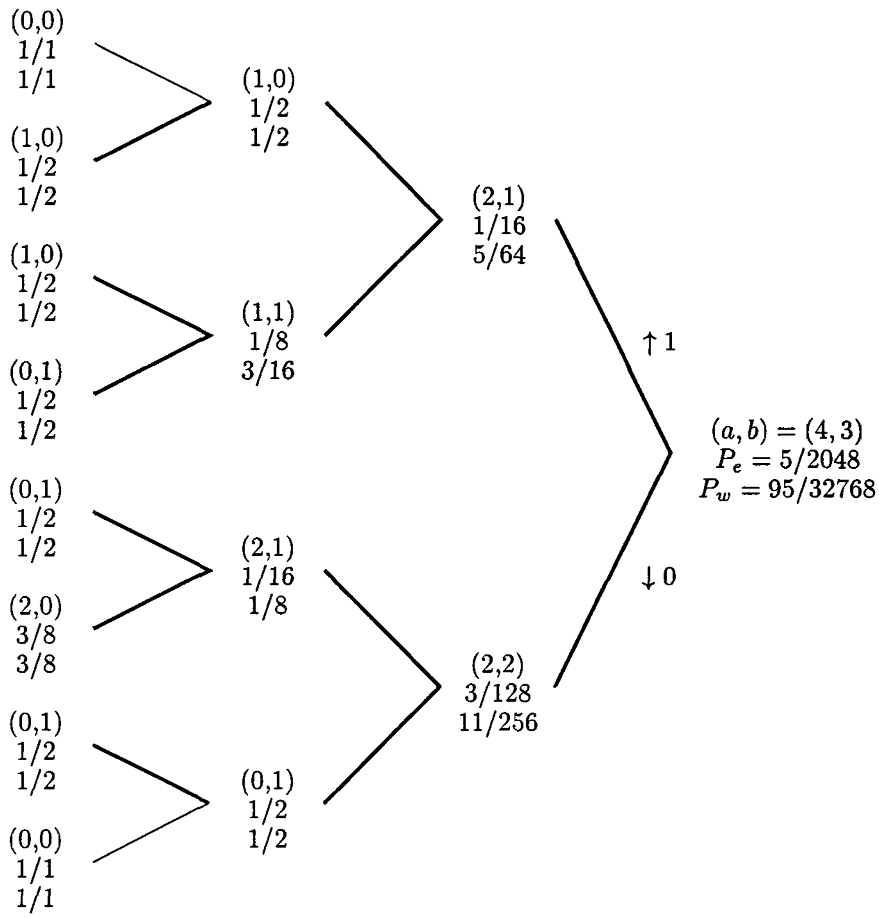


FIGURE 3.2. Weighted context tree $\mathcal{T}_3$ for $x_1^T = 0110100$ and $x_{1-D}^0 = \cdots 010$.

Using (14) we obtain the following upper bound for the model redundancy :

$$\log \frac{\prod_{s \in \mathcal{S}} P_e(a_s, b_s)}{P_c(x_1^T | x_{1-D}^0)} \leq \log \frac{1}{2^{-\Gamma_D(\mathcal{S})}} = \Gamma_D(\mathcal{S}). \tag{23}$$

Combining (20), (21), and (23) in (19) yields the theorem. □

Theorem 6.3 is the basic result in this paper. In this theorem we recognize beside the coding and parameter redundancy the model redundancy. Model redundancy is a consequence of not knowing the (actual, or best in the sense of minimizing (18)) model $\mathcal{S}$, and therefore not being able to take distribution $\prod_{s \in \mathcal{S}} P_e(a_s, b_s)$ as coding distribution. This results in a loss, the model redundancy, which is upper bounded by $\Gamma_D(\mathcal{S})$ bits. Note that in section 2 we have described a natural code that would need $\Gamma_D(\mathcal{S})$ bits to specify the model $\mathcal{S}$. Therefore our weighted method is at least as good as a two-pass method, in which first the best model is determined and transmitted, followed by the code for the sequence given that model.

EXAMPLE 6.1. Suppose a source generated the sequence $x_1^T = 0110100$ with sequence of past symbols $x_{1-D}^0 = \cdots 010$. For $D = 3$ we have plotted the weighted context tree $\mathcal{T}_D$ in Figure 3.2. Node s contains the counts (a_s, b_s), the Krichevsky-Trofimov estimate

$P_e(a_s, b_s)$, and the weighted probability P_w^s. The coding probability corresponding to this sequence is 95/32768.

The upper bound for the model redundancy with respect to the model $\mathcal{S} = \{00, 10, 1\}$ of the source from example 2.1 is $\Gamma_D(\mathcal{S}) = 5$ bits. This also follows quite easily from

$$\begin{aligned} P_w^\lambda &\geq \frac{1}{2}P_w^0 P_w^1 \geq \frac{1}{2}\left(\frac{1}{2}P_w^{00}P_w^{10}\right)P_w^1 \\ &\geq \frac{1}{2}\left(\frac{1}{2}\left(\frac{1}{2}P_e(a_{00}, b_{00})\right)\left(\frac{1}{2}P_e(a_{10}, b_{10})\right)\right)\left(\frac{1}{2}P_e(a_1, b_1)\right). \end{aligned} \tag{24}$$

6.3. Implementation of the context tree weighting method. Before discussing implementation issues we refer to appendix 9 for notation concerning arithmetic encoding and decoding.

6.3.1. *Encoding.* First we set $B(\phi|x_{1-D}^0) := 0$. Then, for $t = 1, 2, \cdots, T$ we create nodes $s(d) := x_{t-d}^{t-1}$ for $d = 0, 1, \cdots, D$ (if necessary), we do a dummy 0-update on these nodes to find $P_c(x_1^{t-1}, X_t = 0|x_{1-D}^0)$, we update $B(x_1^{t-1}|x_{1-D}^0)$ to $B(x_1^{t-1}, X_t = x_t|x_{1-D}^0)$, and we then do the actual update of the nodes $s(d)$ for $d = 0, 1, \cdots, D$ for $X_t = x_t$. This results in $P_c(x_1^{t-1}, X_t = x_t|x_{1-D}^0)$. After having processed $x_1x_2\cdots x_T$ we compute $c^L(x_1^T|x_{1-D}^0)$ from $B(x_1^T|x_{1-D}^0)$ and $P_c(x_1^T|x_{1-D}^0)$.

6.3.2. *Decoding.* First we set $B(\phi|x_{1-D}^0) := 0$ and determine F_∞ from $c_1c_2\cdots c_{L(x_1^T)}$ $c_{L(x_1^T)+1}, \cdots$. Then, for $t = 1, 2, \cdots, T$ we create nodes $s(d) := x_{t-d}^{t-1}$ for $d = 0, 1, \cdots, D$ (if necessary), we do a dummy 0-update on these nodes to find $P_c(x_1^{t-1}, X_t = 0|x_{1-D}^0)$, we compare F_∞ with $B(x_1^{t-1}|x_{1-D}^0) + P_c(x_1^{t-1}, X_t = 0|x_{1-D}^0)$ to find x_t, update $B(x_1^{t-1}|x_{1-D}^0)$ to $B(x_1^{t-1}, X_t = x_t|x_{1-D}^0)$, and we then do the actual update of the nodes $s(d)$ for $d = 0, 1, \cdots, D$ for $X_t = x_t$. This yields $P_c(x_1^{t-1}, X_t = x_t)$. After having processed $x_1x_2\cdots x_T$ we compute $L(x_1^T)$ from $P_c(x_1^T)$ so that we know the start of the next codeword.

6.3.3. *Comments.* We assume that a node $s \in \mathcal{T}_D$ contains the pair (a_s, b_s), the estimated probability $P_e(a_s, b_s)$ and the weighted probability P_w^s. When a node is created, the counts a_s and b_s are made 0, the probabilities $P_e(a_s, b_s)$ and P_w^s are made 1.

Doing a dummy 0-update of the nodes $s(d)$ for $d = 0, 1, \cdots, D$ means that we assume that $X_t = 0$. Then, for $d = D, D-1, \cdots, 0$ we update as indicated by (9) in lemma 5.1,

$$\tilde{P}_e(a_{s(d)} + 1, b_{s(d)}) := P_e(a_{s(d)}, b_{s(d)}) \cdot \frac{a_{s(d)} + 1/2}{a_{s(d)} + b_{s(d)} + 1}, \tag{25}$$

where the tilde above a variable indicates that this variable is a temporary one. After that we form $\tilde{P}_w^{s(D)} := \tilde{P}_e(a_{s(D)} + 1, b_{s(D)})$, and for $d = D-1, D-2, \cdots, 0$ we compute

$$\tilde{P}_w^{s(d)} := \frac{1}{2}\tilde{P}_e(a_{s(d)} + 1, b_{s(d)}) + \frac{1}{2}\tilde{P}_w^{x_{t-d-1}s(d)}P_w^{\overline{x}_{t-d-1}s(d)} \tag{26}$$

where we note that $x_{t-d-1}s(d) = s(d+1)$, so $P_w^{s(d+1)}$ was changed, and $P_w^{\overline{x}_{t-d-1}s(d)}$ has remained the same (see lemma 6.2). All this eventually results in $P_c(x_1^{t-1}, X_t = 0|x_{1-D}^0)$.

It will clear from (36) that

$$B(x_1^{t-1}, X_t = x_t|x_{1-D}^0) := \begin{cases} B(x_1^{t-1}|x_{1-D}^0) & \text{if } x_t = 0 \\ B(x_1^{t-1}|x_{1-D}^0) + P_c(x_1^{t-1}, X_t = 0|x_{1-D}^0) & \text{if } x_t = 1. \end{cases}$$

It should be noted that we use block probabilities to feed into the arithmetic encoder and decoder instead of conditional probabilities as usual. This avoids multiplications in the arithmetic encoder and decoder, which is a pleasant side effect of the weighted approach.

If $X_t = 0$, the actual update is identical to (25) and (26), the only difference is that now we update $P_e(a_s, b_s)$ and P_w^s and increment a_s instead of computing the temporary values $\tilde{P}_e(a_s, b_s)$ and $\tilde{P}_w^s$. If $X_t = 1$ the actual update requires incrementing of b_s, etc. Note that we only have to update the nodes $s(d)$ for $d = 0, 1, \cdots, D$, the nodes along the path in the context tree that is determined by the past symbols x_{t-D}^{t-1}.

The codeword $c^L(x_1^T)$ is finally computed as in definition (33) in appendix 9 and transmitted to the decoder.

The decoder forms F_∞ as in (32) in appendix 9. Note that F_∞ is compared to the threshold $D(x_1^{t-1}) = B(x_1^{t-1}|x_{1-D}^0) + P_c(x_1^{t-1}, X_t = 0|x_{1-D}^0)$, see (39) appendix 9. Finally, the length $L(x_1^T)$ is computed as in definition 9.3.

6.3.4. *Complexity issues.* For each symbol x_t we have to visit $D + 1$ nodes. Some of these nodes have to be created first. From this it follows that the total number of allocated nodes can not be more than $T(D + 1)$. This makes the *storage complexity* not more than *linear* in T. Note also that the number of nodes can not be more than $2^{D+1} - 1$, the total number of nodes in $\mathcal{T}_D$. This shows exponential behavior in D. We did not take into account here, that for infinite precision arithmetic, the number of digits that are needed to specify the counts a_s and b_s and the probabilities $P_e(a_s, b_s)$ and P_w^s, is growing with increasing t, making the storage space for one node measured in e.g. bytes getting bigger each time.

The *computational complexity*, i.e. the number of additions, multiplications, and divisions, is proportional to the number of nodes that are visited, which is $T(D+1)$. Therefore this complexity is also *linear* in T. Again we have neglected the fact here, that for infinite precision arithmetic the number of digits that are needed to specify the counts a_s and b_s and the probabilities $P_e(a_s, b_s)$ and P_w^s, is growing rapidly, making additions, multiplications and divisions becoming more complex with increasing t.

7. Other Weightings

The coding distribution defined by (12) (and (14)) yields model cost not more $2|\mathcal{S}| - 1$, i.e. linear in $|\mathcal{S}|$, if we assume that $\mathcal{S}$ has no leaves at depth D. This is achieved by giving equal weight to $P_e(a_s, b_s)$ and $P_w^{0s} P_w^{1s}$ in each (internal) node $s \in \mathcal{T}_D$.

It is very well possible however to assume that these weights are not equal, and even to suppose that they are different for different nodes s. In this section we will assume that the weigthing in a node s depends on the depth $l(s)$ of this node in the context tree $\mathcal{T}_D$. Hence

$$P_w^s \triangleq \alpha_{l(s)} P_e(a_s, b_s) + (1 - \alpha_{l(s)}) P_w^{0s} P_w^{1s}, \text{ with } \alpha_D = 1. \tag{27}$$

Now note that each model can be regarded as the empty (memoryless) model $\{\lambda\}$ to which a number of nodes may have been added. The cost of the empty model is $-\log \alpha_0$, we can also say that the model cost of the first parameter is $-\log \alpha_0$ bits. Our objective is now that, if we add a new node (parameter) to a model, the model cost increases by δ bit, no matter at what level d we add this node. In other words

$$\frac{1-\alpha_d}{\alpha_d}\cdot\alpha_{d+1}^2 = 2^{-\delta}, \tag{28}$$

for $0 \le d \le D-1$, or consequently

$$\left(\frac{1}{\alpha_d}\right) = 2^{-\delta}\left(\frac{1}{\alpha_{d+1}}\right)^2 + 1. \tag{29}$$

If we now assume that $\delta = 0$, which implies that all models that fit into $\mathcal{S} \in \mathcal{C}_D$ have *equal cost*, we find that $(\alpha_{D-1})^{-1} = 2, (\alpha_{D-2})^{-1} = 5, (\alpha_{D-3})^{-1} = 26, (\alpha_{D-4})^{-1} = 677$, etc. This yields a cost of $\log 677 = 9.403$ bit for all 677 models in $\mathcal{T}_4$ and 150.448 bit for $D = 8$, etc. Note that the number of models in $\mathcal{C}_D$ grows very fast with D. Incrementing D by one results roughly in squaring the number of models in $\mathcal{C}_D$. The context tree weighting method is working on all these models simultaneously in a very efficient way!

If we take δ such that $-\log\alpha_0 = \delta$, we obtain model cost $\delta|\mathcal{S}|$, which is *proportional* to the number of parameters $|\mathcal{S}|$. For $D = 1$ we find that $\delta = -\log\frac{\sqrt{5}-1}{2} = 0.694$ bit, for $D = 2$ we get $\delta = 1.047$ bit, $\delta = 1.411$ bit for $D = 4$ and for $D = 8$ we find $\delta = 1.704$ bit, etc.

8. Final Remarks

We have seen in lemma 6.1 that $P_c(x_1^T|x_{1-D}^0)$ as given by (14) is a weighting over all distributions $\prod_{s\in\mathcal{S}} P_e(a_s, b_s)$ corresponding to models $\mathcal{S} \in \mathcal{C}_D$. From (8) we may conclude that $\prod_{s\in\mathcal{S}} P_e(a_s, b_s)$ is a weighting of $\prod_{s\in\mathcal{S}}(1-\theta_s)^{a_s}\theta_s^{b_s}$, where all components of $\Theta_{\mathcal{S}}$ are assumed to be $(\frac{1}{2},\frac{1}{2})$-Dirichlet distributed. Therefore we may say that $P_c(x_1^T|x_{1-D}^0)$ is a weighting over all models $\mathcal{S} \in \mathcal{C}_D$ and all parameter vectors $\Theta_{\mathcal{S}}$, also called a "double mixture" (see [**47**]). We should stress that the context tree weighting method induces a certain weighting over all models (see lemma 6.1), which can be changed as e.g. in section 7 in order to achieve specific model redundancy behavior.

The redundancy upper bound in theorem 6.3 shows that our method achieves the lower bound obtained by Rissanen (see e.g. [**26**], theorem 1) for finite state sources. However our redundancy bound is in fact stronger, since it holds for all source sequences x_1^T given x_{1-D}^0 and all T, and not only averaged over all source sequences x_1^T given x_{1-D}^0 only for T large enough. Our bound is also stronger in the sense that it is more precise about the terms that tell us about the model redundancy.

The context tree weighting procedure was presented first at the 1993 IEEE International Symposium on Information Theory in San Antonio, Texas (see [**56**]). At the same time Weinberger, Rissanen and Feder[48] studied finite memory tree sources and proposed a method that is based on state estimation. Again an (artificial) constant C and a function $g(t)$ was needed to regulate the selection process. Although we claim that the context tree method has eliminated all these artificial parameters we must admit that the basic context tree method, which is described here, has D as a parameter to be specified in advance, making the method work only for models $\mathcal{S} \in \mathcal{C}_D$, i.e. for models with memory not larger than D. It is however possible (see [**52**]) to modify the algorithm such that there is no constraint on the maximum memory depth D involved (Moreover it was demonstrated there that it is not necessary to have access to x_{1-D}^0.) This implementation thus realizes *infinite* context tree depth D. The storage complexity still remains linear in T. It was furthermore shown in [**52**] that this implementation of the context tree weighting method achieves *entropy* for *any* stationary and ergodic source.

In a recent paper Weinberger, Merhav and Feder[**47**] consider the model class containing the finite state sources (and not only the bounded memory tree sources). They strengthened the Shtarkov pointwise minimax lower bound on the individual redundancy ([**37**], theorem 1), i.e. they found a lower bound (equivalent to Rissanen's lower bound for average redundancy [**26**]) that holds for most sequences in most types. Moreover they investigated the weighted ("mixing") approach for finite state sources. Weinberger et al. showed that the redundancy for the weighted method achieves their strong lower bound. Furthermore their paper shows by an example that the state-estimation approach, the authors call this the "plug-in" approach, does not work for all source sequences, i.e. does not achieve the lower bound.

Finite accuracy implementations of the context tree weighting method in combination with arithmetic coding are studied in [**54**]. In [**58**] context weighting methods are described that perform on more general model classes than the one that we have studied here. These model classes are still bounded memory, and the proposed schemes for them are constructive just like the context tree weighting method that is described here.

Although we have considered only binary sources here, there exist straightforward generalizations of the context tree weighting method to non-binary sources (see e.g. [**43**]).

Acknowledgement

This research was carried out in May 1992 while the second author visited the Information Theory Group at Eindhoven University. The authors thank the Eindhovens Universiteitsfonds for supporting this visit.

Paul Volf participated in research connected to section 7. Thanks. The comments from the reviewers and the advice of the associate editor Meir Feder are also acknowledged here.

9. Appendix: Elias Algorithm

The first idea behind the Elias algorithm is that to each source sequence x_1^T there corresponds a *subinterval* of $[0,1)$. This principle can be traced back to Shannon[**36**].

DEFINITION 9.1. The interval $I(x_1^t)$ corresponding to $x_1^t \in \{0,1\}^t, t = 0,1,\cdots,T$ is defined as

$$I(x_1^t) \triangleq \left[B(x_1^t), B(x_1^t) + P_c(x_1^t)\right) \tag{30}$$

where $B(x_1^t) \triangleq \sum_{\tilde{x}_1^t < x_1^t} P_c(\tilde{x}_1^t)$ for some ordering over $\{0,1\}^t$.

Note that for $t = 0$ we have that $P_c(\phi) = 1$ and $B(\phi) = 0$ (the only sequence of length 0 is ϕ itself), and consequently $I(\phi) = [0,1)$. Observe that for any fixed value of $t, t = 0,1,\cdots,T$, all intervals $I(x_1^t)$ are disjoint, and their union is $[0,1)$. Each interval has a length equal to the corresponding coding probability.

Just like all source sequences, a codeword $c^L = c_1c_2\cdots c_L$ can be associated with a subinterval of $[0,1)$.

DEFINITION 9.2. The interval $J(c^L)$ corresponding to the codeword c^L is defined as

$$J(c^L) \triangleq \left[F(c^L), F(c^L) + 2^{-L}\right), \tag{31}$$

with $F(c^L) \triangleq \sum_{l=1,L} c_l 2^{-l}$.

To understand this, note that c^L can be considered as a binary fraction $F(c^L)$. Since c^L is followed by other codewords, the decoder receives a stream of code digits from which only the first L digits correspond to c^L. The decoder can determine the value that is represented by the binary fraction formed by the total stream $c_1c_2\cdots c_Lc_{L+1}\cdots$, i.e.

$$F_\infty \triangleq \sum_{l=1,\infty} c_l 2^{-l}, \tag{32}$$

where it should be noted that the length of the total stream is not necessarily infinite. Since $F(c^L) \leq F_\infty < F(c^L) + 2^{-L}$ we may say that the interval $J(c^L)$ corresponds to the codeword c^L.

To compress a sequence x_1^T, we search for a (short) codeword $c^L(x_1^T)$ whose code interval $J(c^L)$ is contained in the sequence interval $I(x_1^T)$.

DEFINITION 9.3. The codeword $c^L(x_1^T)$ for source sequence x_1^T consists of $L(x_1^T) \triangleq \lceil \log(1/P_c(x_1^T)) \rceil + 1$ binary digits such that

$$F(c^L(x_1^T)) \triangleq \lceil B(x_1^T) \cdot 2^{L(x_1^T)} \rceil \cdot 2^{-L(x_1^T)}, \tag{33}$$

where $\lceil a \rceil$ is the smallest integer $\geq a$. We consider only sequences x_1^T with $P_c(x_1^T) > 0$.

Since

$$F(c^L(x_1^T)) \geq B(x_1^T) \tag{34}$$

and

$$F(c^L(x_1^T)) + 2^{-L(x_1^T)} < B(x_1^T) + 2^{-L(x_1^T)} + 2^{-L(x_1^T)} \leq B(x_1^T) + P_c(x_1^T). \tag{35}$$

we may conclude that $J(c^L(x_1^T)) \subseteq I(x_1^T)$, and therefore $F_\infty \in I(x_1^T)$. Since all intervals $I(x_1^T)$ are disjoint, the decoder can reconstruct the source sequence x_1^T from F_∞. Note that after this reconstruction, the decoder can compute $c^L(x_1^T)$, just like the encoder, and find the location of the first digit of the next codeword. Note also that, since all code intervals are disjoint, no codeword is the prefix of any other codeword. This implies also that the code satisfies the prefix condition. From the definition of the length $L(x_1^T)$ we immediately obtain theorem 4.1.

The second idea behind the Elias algorithm, is to order the sequences x_1^t of length t *lexicographically*, for $t, t = 1, \cdots, T$. For two sequences x_1^t and $\tilde{x}_1^t$ we have that $x_1^t < \tilde{x}_1^t$ if and only if there exists a $\tau \in \{1, 2, \cdots, t\}$ such that $x_i = \tilde{x}_i$ for $i = 1, 2, \cdots, \tau - 1$ and $x_\tau < \tilde{x}_\tau$. This lexicographical ordering makes it possible to compute the interval $I(x_1^T)$ sequentially. To do so, we transform the starting interval $I(\phi) = [0, 1)$ into $I(x_1), I(x_1x_2)$, $\cdots$, and $I(x_1x_2\cdots x_T)$ respectively. The consequence of the lexicographical ordering over the source sequences is that

$$B(x_1^t) = \sum_{\tilde{x}_1^t < x_1^t} P_c(\tilde{x}_1^t) = \sum_{\tilde{x}_1^{t-1} < x_1^{t-1}} P_c(\tilde{x}_1^{t-1}) + \sum_{\tilde{x}_t < x_t} P_c(x_1^{t-1}, \tilde{x}_t) = B(x_1^{t-1}) + \sum_{\tilde{x}_t < x_t} P_c(x_1^{t-1}, \tilde{x}_t). \tag{36}$$

In other words $B(x_1^t)$ can be computed from $B(x_1^{t-1})$ and $P_c(x_1^{t-1}, X_t = 0)$. Therefore the encoder and the decoder can easily find $I(x_1^t)$ after having determined $I(x_1^{t-1})$, if it is 'easy' to determine probabilities $P_c(x_1^{t-1}, X_t = 0)$ and $P_c(x_1^{t-1}, X_t = 1)$ after having processed $x_1x_2\cdots x_{t-1}$.

Observe that when the symbol x_t is being processed, the source interval $I(x_1^{t-1}) = [B(x_1^{t-1}), B(x_1^{t-1}) + P_c(x_1^{t-1}))$ is subdivided into two subintervals

$$\begin{aligned} I(x_1^{t-1}, X_t = 0) &= \left[B(x_1^{t-1}), B(x_1^{t-1}) + P_c(x_1^{t-1}, X_t = 0)\right) \text{ and} \\ I(x_1^{t-1}, X_t = 1) &= \left[B(x_1^{t-1}) + P_c(x_1^{t-1}, X_t = 0), B(x_1^{t-1}) + P_c(x_1^{t-1})\right). \end{aligned} \tag{37}$$

The encoder proceeds with one of these subintervals depending on the symbol x_t, therefore $I(x_1^t) \subseteq I(x_1^{t-1})$. This implies that

$$I(x_1^T) \subseteq I(x_1^{T-1}) \subseteq \cdots \subseteq I(\phi). \tag{38}$$

The decoder determines from F_∞ the source symbols $x_1, x_2, \cdots, x_T$ respectively by comparing F_∞ to thresholds $D(x_1^{t-1})$.

DEFINITION 9.4. The thresholds $D(x_1^{t-1})$ are defined as

$$D(x_1^{t-1}) \triangleq B(x_1^{t-1}) + P_c(x_1^{t-1}, X_t = 0), \tag{39}$$

for $t = 1, 2, \cdots, T$.

Observe that threshold $D(x_1^{t-1})$ splits up the interval $I(x_1^{t-1})$ in two parts (see (37)). It is the upper boundary point of $I(x_1^{t-1}, X_t = 0)$ but also the lower boundary point of $I(x_1^{t-1}, X_t = 1)$. Since always $F_\infty \in I(x_1^T) \subseteq I(x_1^t)$, we have for $D(x_1^{t-1})$ that

$$F_\infty < B(x_1^t) + P_c(x_1^t) = B(x_1^{t-1}) + P_c(x_1^{t-1}, X_t = 0) = D(x_1^{t-1}) \text{ if } x_t = 0, \tag{40}$$

and

$$F_\infty \geq B(x_1^t) = B(x_1^{t-1}) + P_c(x_1^{t-1}, X_t = 0\} = D(x_1^{t-1}) \text{ if } x_t = 1. \tag{41}$$

Therefore the decoder can easily find x_t by comparing F_∞ to the threshold $D(x_1^{t-1})$, in other words it can operate sequentially.

Since the code satisfies the prefix condition, it should not be necessary to have access to the complete F_∞ for decoding x_1^T. Indeed, it can be shown that only the first $L(x_1^T)$ digits of the codestream are actually needed.

10. Appendix: Properties of the KT-Estimator

PROOF. The proof consists of two parts.

1. The fact that $P_e(0,0) = 1$ follows from

$$\int_0^1 \frac{1}{\sqrt{\theta(1-\theta)}} d\theta = \int_0^{\pi/2} \frac{1}{\sin\alpha\cos\alpha} d\sin^2\alpha = \int_0^{\pi/2} 2d\alpha = \pi. \tag{42}$$

It is easy to see that $P_e(a+1,b) + P_e(a,b+1) = P_e(a,b)$. We obtain (9) from

$$\begin{aligned}(b+1/2)P_e(a+1,b) &= \frac{b+1/2}{\pi}\int_0^1 (1-\theta)^{a+1/2}\theta^{b-1/2}d\theta \\ &= \frac{1}{\pi}\int_0^1 (1-\theta)^{a+1/2}d\theta^{b+1/2} \\ &= -\frac{1}{\pi}\int_0^1 \theta^{b+1/2}d(1-\theta)^{a+1/2} \\ &= \frac{a+1/2}{\pi}\int_0^1 (1-\theta)^{a-1/2}\theta^{b+1/2}d\theta = (a+1/2)P_e(a,b+1). \end{aligned}\tag{43}$$

2. Define

$$\Delta(a,b) \triangleq \frac{P_e(a,b)}{\frac{1}{\sqrt{a+b}}(\frac{a}{a+b})^a(\frac{b}{a+b})^b}. \tag{44}$$

First we assume that $a \geq 1$. Consider

$$\frac{\Delta(a+1,b)}{\Delta(a,b)} = \frac{a^a(a+1/2)}{(a+1)^{a+1}} \cdot (\frac{a+b+1}{a+b})^{a+b+1/2}. \tag{45}$$

To analyze (45) we define, for $t \in [1,\infty)$, the functions

$$f(t) \triangleq \ln\frac{t^t(t+1/2)}{(t+1)^{t+1}} \quad \text{and} \quad g(t) \triangleq \ln(\frac{t+1}{t})^{t+1/2}. \tag{46}$$

The derivatives of these functions are

$$\frac{df(t)}{dt} = \ln\frac{t}{t+1} + \frac{1}{t+1/2} \quad \text{and} \quad \frac{dg(t)}{dt} = \ln\frac{t+1}{t} - \frac{t+1/2}{t(t+1)}. \tag{47}$$

Take $\alpha = \frac{1/2}{t+1/2}$ and observe that $0 < \alpha \leq 1/3$. Then from

$$\ln\frac{t}{t+1} = \ln\frac{1-\alpha}{1+\alpha} = -2(\alpha + \frac{\alpha^3}{3} + \frac{\alpha^5}{5} + \cdots) \leq -2\alpha = -\frac{1}{t+1/2}, \tag{48}$$

we obtain that $\frac{df(t)}{dt} \leq 0$. Therefore

$$\frac{a^a(a+1/2)}{(a+1)^{a+1}} \geq \lim_{a\to\infty} \frac{a^a(a+1/2)}{(a+1)^{a+1}} = \frac{1}{e}. \tag{49}$$

Similarly from

$$\ln\frac{t+1}{t} = 2(\alpha + \frac{\alpha^3}{3} + \frac{\alpha^5}{5} + \cdots) \leq 2(\alpha + \alpha^3 + \alpha^5 + \cdots) = \frac{2\alpha}{1-\alpha^2} = \frac{t+1/2}{t(t+1)}, \tag{50}$$

we may conclude that $\frac{dg(t)}{dt} \leq 0$. This results in

$$(\frac{a+b+1}{a+b})^{a+b+1/2} \geq \lim_{a+b\to\infty}(\frac{a+b+1}{a+b})^{a+b+1/2} = e. \tag{51}$$

Combining (49) and (51) yields that

$$\Delta(a+1,b) \geq \Delta(a,b) \text{ for } a \geq 1. \tag{52}$$

Next we investigate the case where $a = 0$. Note that this implies that $b \geq 1$, and consider

$$\frac{\Delta(1,b)}{\Delta(0,b)} = \frac{1}{2}\cdot(\frac{b+1}{b})^{b+1/2}. \tag{53}$$

If we again use the fact that $\frac{dg(t)}{dt} \leq 0$, we find that

$$\Delta(1,b) \geq \frac{e}{2}\cdot\Delta(0,b). \tag{54}$$

Inequality (52) together with (54), now implies that

$$\Delta(a+1,b) \geq \Delta(a,b) \text{ for } a \geq 0. \tag{55}$$

Therefore

$$\Delta(a,b) \geq \Delta(0,1) = \Delta(1,0). \tag{56}$$

The lemma now follows from the observation $\Delta(1,0) = \Delta(0,1) = 1/2$. It can also easily be proved that $\Delta(a,b) \leq \sqrt{2/\pi}$. Both bounds are tight. □

11. Appendix: Weighting Properties

PROOF. We prove by induction that the hypothesis in lemma 6.1 holds. For $d = D$ this is true. Next assume that the hypothesis also holds for $0 < d \leq D$. Now consider a node s with $l(s) = d-1$, then

$$\begin{aligned}
P_w^s &= \frac{1}{2}P_e(a_s,b_s) + \frac{1}{2}P_w^{0s}\cdot P_w^{1s} \\
&= \frac{1}{2}P_e(a_s,b_s) \\
&\quad + \frac{1}{2}\left(\sum_{\mathcal{V}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{V})}\prod_{v\in\mathcal{V}} P_e(a_{v0s},b_{v0s})\right)\left(\sum_{\mathcal{W}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{W})}\prod_{w\in\mathcal{W}} P_e(a_{w1s},b_{w1s})\right) \\
&= 2^{-1}P_e(a_s,b_s) + \sum_{\mathcal{V},\mathcal{W}\in\mathcal{C}_{D-d}} 2^{-1-\Gamma_{D-d}(\mathcal{V})-\Gamma_{D-d}(\mathcal{W})} \prod_{u\in\mathcal{V}\times 0\cup\mathcal{W}\times 1} P_e(a_{us},b_{us}) \\
&= \sum_{\mathcal{U}\in\mathcal{C}_{D-d+1}} 2^{-\Gamma_{D-d+1}(\mathcal{U})}\prod_{u\in\mathcal{U}} P_e(a_{us},b_{us}).
\end{aligned} \tag{57}$$

We have used the induction hypothesis in the second step of the derivation. Conclusion is that the hypothesis also holds for $d-1$, and by induction for all $0 \le d \le D$.
The fact $\sum_{\mathcal{U}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} = 1$ can be proved similarly if we note that

$$\sum_{\mathcal{U}\in\mathcal{C}_{D-d+1}} 2^{-\Gamma_{D-d+1}(\mathcal{U})} = \frac{1}{2} + \frac{1}{2}\left(\sum_{\mathcal{V}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{V})}\right)\cdot\left(\sum_{\mathcal{W}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{W})}\right) = \frac{1}{2}+\frac{1}{2} = 1. \tag{58}$$

□

12. Appendix: Updating Properties

PROOF. First note that if s is not a suffix of x_{t-D}^{t-1} no descendant of s can be suffix of x_{t-D}^{t-1}. Therefore for s and its descendants the a- and b-counts remain the same, and consequently also the estimated probabilities $P_e(a_s,b_s)$, after having observed the symbol x_t. This implies that also the weighted probability P_w^s does not change and (15) holds.

For those $s \in \mathcal{T}_D$ that are a suffix of x_{t-D}^{t-1} we will show that the hypothesis (16) holds by induction. Observe that (16) holds for $l(s) = D$. To see this note that for s such that $l(s) = D$

$$P_w^s(0) + P_w^s(1) = P_e(a_s+1,b_s) + P_e(a_s,b_s+1) = P_e(a_s,b_s) = P_w^s(\phi). \tag{59}$$

(Notation : $P_w^s(0) = P_w^s(x_1^{t-1}, X_t = 0|x_{1-D}^0)$, $P_w^s(1) = P_w^s(x_1^{t-1}, X_t = 1|x_{1-D}^0)$, $P_w^s(\phi) = P_w^s(x_1^{t-1}|x_{1-D}^0)$, $a_s = a_s(x_1^{t-1}|x_{1-D}^0)$, $b_s = b_s(x_1^{t-1}|x_{1-D}^0)$.)
Next assume that (16) holds for $l(s) = d, 0 < d \le D$. Now consider nodes corresponding to strings s with $l(s) = d-1$. Then $1s$ is a postfix of x_{t-D}^{t-1} and $0s$ not, or vice versa. Let

$1s$ be a postfix of x_{t-D}^{t-1}, then

$$\begin{aligned}
P_w^s(0) + P_w^s(1) &= \frac{1}{2}P_e(a_s+1,b_s) + \frac{1}{2}P_w^{0s}(0)\cdot P_w^{1s}(0) + \frac{1}{2}P_e(a_s,b_s+1) + \frac{1}{2}P_w^{0s}(1)\cdot P_w^{1s}(1) \\
&= \frac{1}{2}P_e(a_s+1,b_s) + \frac{1}{2}P_e(a_s,b_s+1) + \frac{1}{2}P_w^{0s}(\phi)\cdot P_w^{1s}(0) + \frac{1}{2}P_w^{0s}(\phi)\cdot P_w^{1s}(1) \\
&= \frac{1}{2}P_e(a_s,b_s) + \frac{1}{2}P_w^{0s}(\phi)\cdot\left(P_w^{1s}(0) + P_w^{1s}(1)\right) \\
&= \frac{1}{2}P_e(a_s,b_s) + \frac{1}{2}P_w^{0s}(\phi)\cdot P_w^{1s}(\phi) = P_w^s(\phi). \qquad (60)
\end{aligned}$$

The induction hypothesis is used to obtain the fourth equality. The second equality follows from (15). The proof is analogous when $0s$ is a postfix of x_{t-D}^{t-1} instead of $1s$. □

Part 2

Description of the Proposed Implementation

CHAPTER 4

Context-Tree Implementation

1. Tree-Implementation in CTW-1

We shall repeat briefly the implementation of the context-tree in the CTW-1 algorithm and we highlight the most important features. In the next section we then discuss the considerations that led to the CTW-2 implementation.

1.1. Binary decomposition. The contex tree weighting method is originally described for binary (tree) sources. One of the most important applications of noiseless source coding algorithms is however compaction of computer files and these files can be regarded as sequences of bytes, a byte being a symbol that can assume 256 values. In CTW-1 we adapted the context-tree weighting method such that it can be used to compress files of bytes. This was realized by visualizing a byte as a sequence of 8 binary digits. We should note that the first binary digit in a byte (the most significant one) has a different statistical structure than the second binary digit in a byte etc. This difficulty can easily be overcome if we use a context tree for each binary digit, so there is a context tree for binary digit 1 (the most significant bit), a context tree for digit 2, etc., eight trees in total. The context for the first bit is of course equal to the most recent, say B bytes, i.e. $8B$ binary digits, since we again consider these bytes as sequences of 8 bits. The context for the second bit in a byte is in addition to the B most recent bytes, the most recent bit which is bit 1 from the current byte, which is assumed to be processed already. The context for bit 3 in the current byte is bit 1 and bit 2 from the current byte followed by the $8B$ bits that form the B most recent bytes.

1.2. Weighting only at byte-boundaries. In Figure 4.1 we have depicted the decomposition of the bytes into binary digits and we have shown how the context for each of these digits looks like. This approach more or less suggests that for each of the eight binary digits in a byte there exists a *binary* tree model that describes the statistical structure of the subsequence formed by only these digits. If we apply the context tree weighting procedure directly, these models can be arbitrary (complete) binary trees. However this is a bit strange, it more or less suggests that a certain digit can "depend" on an arbitrary number of binary digits, instead of on an arbitrary number of bytes. This last alternative would be more natural. An additional advantage behind this alternative (by which we make the model-class smaller) is that the number of bits needed to describe a "byte-tree model" is significantly smaller than the number of bits needed to describe the same model as a "bit-tree model". We can achieve this objective by weighting in a binary context tree only at so called byte-boundaries.

In the Figure 4.1 we have visualized this weighting only at byte-boundaries by depicting the context tree as a tree in which we only have nodes that correspond to complete-byte contexts. These nodes have (in principle) 256 children. An exception is formed by almost all the root nodes. The root node in the context tree for bit 1 has 256 children and in this node we do a weighting operation since this root node corresponds to a byte-complete context. However the root node of the context tree for digits $d = 2, 3, \cdots, 8$ has only 2^{d-1} children and in such a root note we only multiply the probabilities at the first byte boundary and we do not weight (this root node is not on a byte-boundary, so it doesn't have its own estimator).

1.3. Binary search trees. Each node in the context tree in Figure 4.1 in principle has 256 possible descendants. If we implement such a structure in software we need storage for 256 pointers to these descendants. Since most of the pointers will be `nil` (no such descendant) this is not very efficient. A better way to implement this is to use a binary tree. Each node in a binary tree corresponds to a 256-ary symbol value and has a left and a right pointer. The left pointer points at nodes with a smaller symbol value, the right pointer to the nodes that have higher symbol values, see Figure 4.2. When a new symbol occurs, a node is added at the correct place in the search tree. The location of this node is found by comparing the new symbol to a current node symbol and taking the left or right direction according to the outcome of this comparison until an endpoint is reached. This implementation has the advantage that the number of pointers is considerably smaller than for a 256-ary tree. The search complexity however is larger.

Although we implemented the algoritm with binary search trees it is sometimes better to think of it as were it a 256-ary tree implementation.

1.4. Pruning of unique context paths. Consider the 256-ary context tree structure. Suppose that the maximum depth is B bytes. At first sight it is reasonable to add

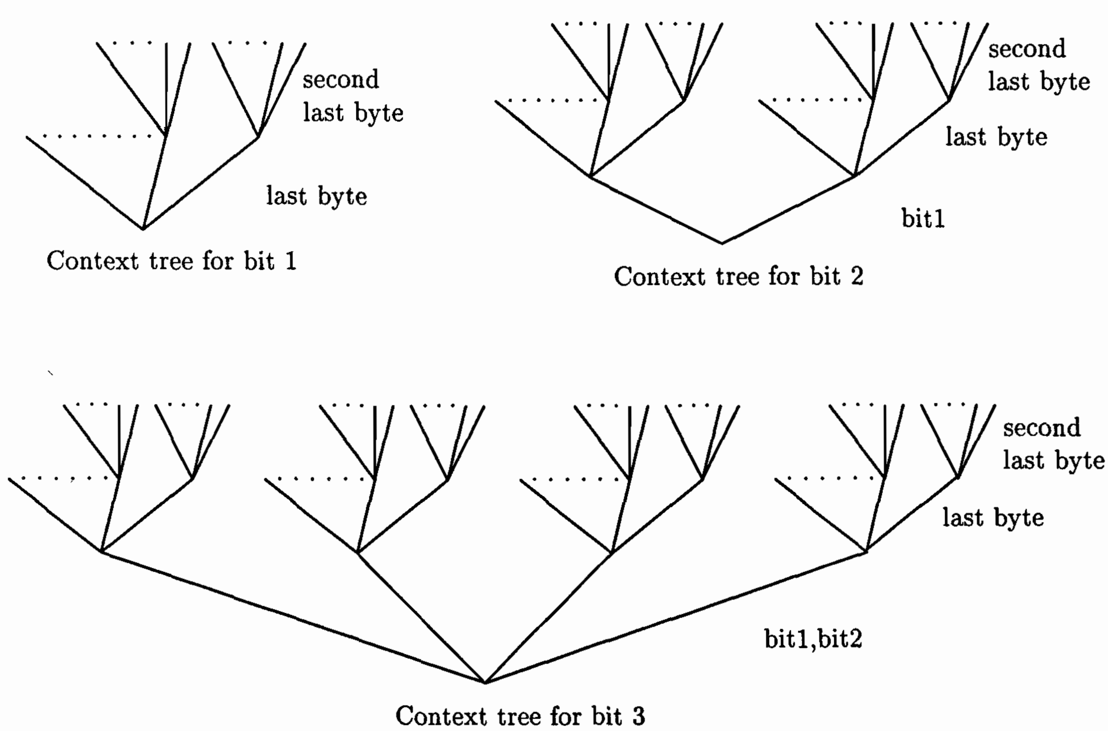


FIGURE 4.1. Decomposition of bytes into binary digits and context descriptions.

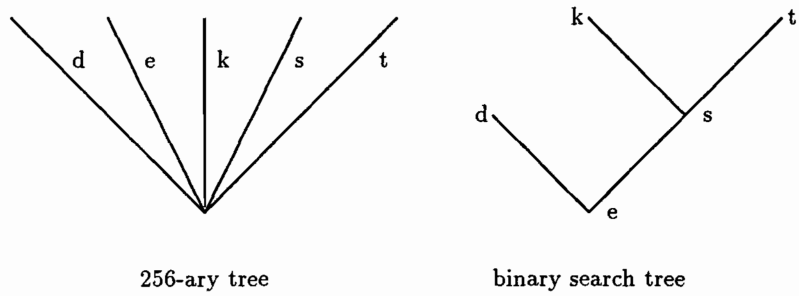


FIGURE 4.2. Binary search tree versus 256-ary tree.

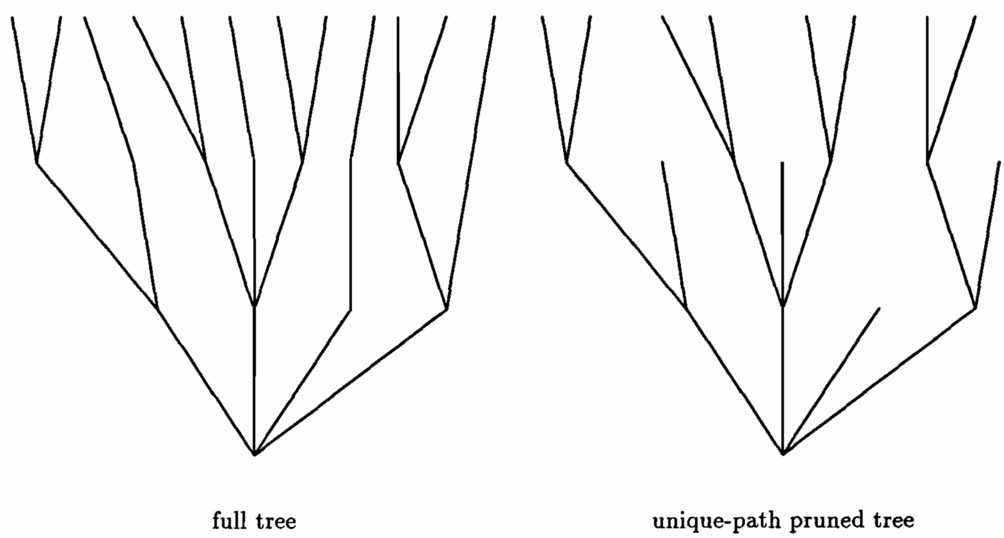


FIGURE 4.3. Binary search tree versus 256-ary tree.

all contexts of length B that have occurred to this structure. This would yield a linear increasing storage complexity. However it is not so useful to continue a path that does not branch anymore. Since in such a path the weighted probability is equal to the estimated probability it is useless to increase the length of the path if it is clear that it does not branch anymore. A problem arises however if at a later time a context occurs that coincides with such a pruned path. Then it is necessary to be able to reconstruct the full (i.e. up to length B) context corresponding to this pruned path. This full context is then compared with the new context and both of them are continued as far as needed to make them unique again. This idea decreases the number of records quite drastically but also makes the implementation more complex, see Figure 4.3. It is necessary to keep the complete past source sequence in memory to be able to reconstruct a pruned context. The amount of memory needed for storage of this sequence is small compared to the decrease of storage requirements due to the pruning in the context tree.

1.5. Memory requirements for a tree node. Each tree node, internal or leaf, contains structure and CTW information. The required CTW information, needed to perform the probabilitiy estimation and weighting, is discussed elsewhere in this report. Here we consider the cost of maintaining the tree structure. The CTW-1 tree is implemented using binary search trees, so we require a left and a right pointer at each node. Also a pointer to the next level, i.e. the context one deeper than the current one, is needed. To facilitate the unique path pruning we furthermore require an index into the file buffer.

The total memory requirements for these four items is 16 bytes; four byte per pointer and four bytes for the index.

2. Tree-Implementation in CTW-2

In CTW-2 we use most of the ideas from CTW-1, i.e. binary decomposition, weighting at byte boundaries, and tree pruning. The main difference between the CTW-1 and the CTW-2 tree implementation is the use of hashing in stead of pointers.

2.1. Binary decomposition and weighting at byte boundaries. We still employ eight separate trees, one for each bit in a byte. These trees are now really byte oriented, i.e. we don't employ binary search trees anymore.

Also, we do a full weighting at the roots of the eight trees. As argued above, this should not be done if we really assume byte oriented data, however the implemetation is more uniform, and thus simpler, while the resulting compression rate and speed is almost the same. In Appendix 4 we show the results of experiments with and without weighting at the root nodes.

2.2. Pruning of unique context paths. Pruning of the unique paths is still implemented as it turned out to reduce the storage requirements considerably. The implementation is the same as in CTW-1. Each node contains an index into the file buffer pointing to the character that caused this node in the tree.

2.3. Memory requirements for a tree node. The structure information per node is now almost only determined by the unique path pruning, so we require an index into the file buffer.

For the hashing implementation we require a hashing table. Since we don't know the sizes of the different trees beforehand we decided to combine the eight trees in one hashing table. This required some extra information in a node; we must know to what tree the node belongs. Usual hashing methods store the key information in the record (node). As we shall see, we can do with only the context depth, because with the unique path pointer we can find the complexe context string.

So, the total memory requirements for this implementation is three bytes for an index and since the tree number and depth information can be combined in one byte, the total memory cost is in four bytes, which is one fourth of the requirements of CTW-1.

3. Hashing of Context Trees

The purpose of a context tree is to supply the CTW information for a given context. So a context tree is actually a data structure that supplies information based on a "context key". This type of information retrieval problems can often be solved efficiently using a hashing table.

In a hashing table, the information (CTW data) is stored and the context is used to find the correct data. The hashing problem can be separated into two sub-problems; the collision resolution scheme and the access or hashing function design. The hashing function assigns to every possible key an index into the hashing table, preferably such that every key is assigned a unique key. Usually however, the key space is much larger than the index space, i.e. the table size. Thus, several keys will be mapped to the same index value. In order to obtain the correct data we will have to resolve these multiple assignments. This is known as the collision resolution problem.

In the following subsections we shall describe the solutions we used in the CTW-2 algorithm. Background information on these topics can be found in [**14**].

3.1. Hashing function. A good hashing function approximates a random assignment of indexes from key values. The key values used here are byte strings of variable length. Pearson [**22**] and Savoy [**33**] discuss this type of hashing function.

Pearson proposes the following hashing function. Let $C[1], C[2], \ldots, C[n]$ be the (byte-string) key and $T[]$ be a pseudo-random permutation of $\{0, \ldots, 255\}$, then the hashing function value is determined by the following program.

```
integer hash(array C) :
  h[0] := 0;
  for i in 1..n loop
    h[i] := T[h[i-1] xor C[i]];
  end loop;
  return h[n];
end proc;
```

This function results in 256 indices.

To obtain a larger index space Pearson suggest to increase the first byte value of the string and apply the function to this string too. Then concatenate the two function values to obtain a 16 bit index.

Savoy applied this function to French texts and concluded that the well known multiplicative scheme performs better in the string case too and suggests the following function.

```
integer hash(array C) :
  h[0] := 0;
  for i in 1..n loop
    h[i] := (h[i-1] * 137 + T[C[i]]) mod 256;
  end loop;
  return h[n];
end proc;
```

3.2. Collision resolution. A well known technique of collision resolution is the open addressing resolution. We choose this method because it works within the preassigned hashtable and doesn't require pointers, as the chaining method would. Because we want to reduce the memory cost we don't accept the cost for pointers.

The same method as Pearson suggest for extending the range of the hashing function can be used to obtain a pseudo random offset value for secondary probes, see Savoy. We must ensure that the offset value is relative prime to the table size, which is always a power of 2. So we obtain the next offset function.

```
integer offset(array C) :
  C[1] := (C[1]+1) mod 256;
  h := hash(C);
  return (h xor 1);
end proc;
```

Thus the complete index computation is performed as in the next function.

```
integer index(array C) :
  h := hash(C);
  if(Table[h] "matches key") then
    return h;
  end if;

  j := offset(C);
  do MAXPROBES times loop
    h := (h + j) mod TABLESIZE;
    if(Table[h] "matches key") then
      return h;
    end if;
  end loop;

  " We haven't found a valid index if we reach this point "
end proc;
```

3.3. Hashing function and Collision resolution for contexts. In our case hashing probes come in sequences of context strings of increasing length x^i, $i = 0, \ldots, d$. Here the index of x^0 is the root index of the tree.

Because the index position of x^i is assumed to be random appearing, we can use this as our first probe for x^{i+1} and thus only compute an offset to probe the table.

```
integer findindex(array C) :
  j := offset(C);
  h := "previous context index";
  do MAXPROBES times loop
    h := (h + j) mod TABLESIZE;
    if(Table[h] "matches key") then
      return h;
    end if;
  end loop;
  return(-1);      /* indicates failure */
end proc;
```

The hash function can be simplified to depend only on the last byte of the context string. This speeds up the hash function enormously, because it can now be represented by a 256 entries integer table costing 1024 bytes. No computations are needed, only one array reference is necessary. The code looks like the following.

```
integer Tperm[256] = { 175715, 11428377, ... , 27394735 };

integer offset(c) :
  return (Tperm[c] and INDMASK);
end proc;
```

These changes increase the speed of the algorithm, while the quality of the hash function, in terms of number of probes needed and the number of failures, remains almost unchanged. In appendix 5 we show the results of the different hashing functions as can be used in the CTW algorithm.

4. Appendix: Weighting of Root Nodes

In this appendix we consider the effects of applying full weighting at root nodes against only using the estimators at the symbol boundaries.

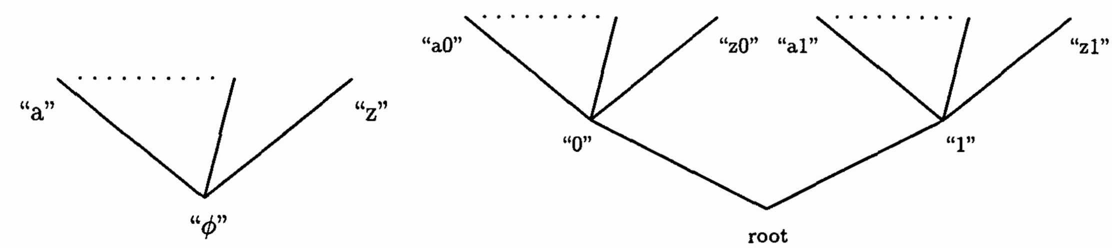


FIGURE 4.4. The tree for bit one and bit two.

First, consider the case when we weight at the root, see Figure 4.4. At the root, λ, we use the estimator $P_e^\lambda(x^t)$ and the weighted probabilities from the nodes "0" and "1" and obtain the weighted probability that we will use as the coding probability. The nodes "0" and "1" do not correspond to a symbol, but represent the first bit of the current byte. In order to have a more uniform program we even implemented this for the first bit, where the prefix has length zero and is represented by the single value "ϕ". Again see Figure 4.4.

$$P_w^\lambda(x^t) = \frac{P_e^\lambda(x^t) + P_w^0(x^t)P_w^1(x^t)}{2} \tag{61}$$

Without full weighting we would use

$$P_w^\lambda(x^t) = P_w^0(x^t)P_w^1(x^t) \tag{62}$$

First, assuming that we are really modeling a symbol source with a 256 symbol alphabet, (62) is the reasonable choice. Assuming that the true source model does not include the root node, (62) produces a codeword about one bit shorter than with (61) per tree. So, the total gain will be about eight bits. On the other hand, if some bits can be modeled by the root node, the extra cost is determined by an excess of parameters, that cost about $\log N$ bits per parameter.

In Table 4.1 we list the results for the CTW algorithm using the K-T estimator and a maximum depth of 4 and 8. Per file from the Calgary corpus the codelength in bits is listed for the CTW with full weighting in the root nodes, together with the codeword length difference when using the CTW without weighting in the root nodes. Also the encoding speed, in Kilobytes per second, of the CTW with weighting in the root is given and also the speed increase in percents resulting from the CTW without rootweighting.

Table 4.2 gives similar results when using the Zero-Redundancy estimator, see 79.

We observe a difference in codelength per file varying from -42 to $+120$ bits. We expected differences from -8 up. This is mostly the case. The occasional lower value (-42) can be explained by the fact that, due to the unique path pruning and the ad-hoc change in program code to allow non-root weighting, the two versions, with and without root weighting, can perform differently in the initial part of the data files. The actual difference strongly depends on these initial parts.

depth	depth 4				depth 8			
file	codelength root weight. bytes	length differ. bytes	coding speed KB/s	speed diff. Δ %	codelength root weight. bytes	length differ. bytes	coding speed KB/s	speed diff. Δ %
bib	220858	-6	6.51	6.30	213614	-8	3.56	3.09
book1	1745933	-13	6.98	5.59	1691669	-5	3.92	3.57
book2	1249714	-7	7.03	5.41	1182291	-7	3.95	3.29
geo	464445	61	6.54	6.12	464319	62	4.78	4.18
news	948138	-7	6.85	6.86	924625	-6	4.14	3.62
obj1	83334	36	5.25	2.48	83423	38	2.36	2.12
obj2	652185	24	7.03	6.54	630041	33	4.21	3.33
paper1	130428	-3	5.97	6.03	129131	-3	3.13	2.24
paper2	192986	-8	6.27	5.74	190607	-6	3.43	2.62
paper3	122127	-6	5.83	5.32	121543	-7	3.01	2.66
paper4	39245	-7	4.05	3.46	39343	-7	1.64	1.22
paper5	36968	-8	3.89	3.60	37002	-7	1.54	1.30
paper6	96840	-7	5.72	4.90	96239	-5	2.80	1.43
pic	418396	8	8.19	8.06	411938	20	4.74	4.22
progc	100126	-9	5.77	3.12	98745	-8	2.84	3.17
progl	133181	-9	6.48	4.78	125177	-5	3.28	2.44
progp	93118	-8	6.10	5.41	89956	-8	2.98	2.35
trans	161033	-8	6.58	6.08	151081	-6	3.43	2.62

TABLE 4.1. The effect of root weighting with the K-T estimator.

The speed increase for the CTW without root weighting, about 5–6 percent for depth 4, is the result of not having to perform about 20 % of the CTW operations. The total time spend in CTW weighting is about 40 %, so we expected an 8 % speed increase. The gain should be somewhat less for depth 8, and the experiments confirm this.

From these results we conclude that weighting or not weighting in the root nodes doesn't much influence the performance of the algorithm. A slightly simpler implementation results when we perform weighting at the root nodes, and this was implemented.

5. Appendix: Performance of Hashing Functions in the CTW Algorithm

In order to determine the performance of the hashing function as proposed by Pearson [**22**], Savoy [**33**], and the CTW hashing function as introduced in section 3.3 we compared the number of probes needed per index computation and the rate of failures to find an index for the function `index` as defined in section 3.2.

depth	depth 4				depth 8			
file	codelength root weight. bytes	length differ. bytes	coding speed KB/s	speed diff. Δ %	codelength root weight. bytes	length differ. bytes	coding speed KB/s	speed diff. Δ %
bib	213084	-5	6.88	3.20	202811	-13	3.77	3.98
book1	1733405	-23	7.06	5.10	1670898	-42	4.15	2.89
book2	1225630	-1	7.14	5.60	1147949	5	4.21	2.85
geo	466110	69	6.67	3.45	465838	63	4.90	4.69
news	923471	-33	7.04	4.40	888626	-40	4.42	3.39
obj1	81266	58	5.38	2.79	81273	60	2.44	2.46
obj2	627289	120	7.22	5.26	596169	110	4.48	4.24
paper1	124775	-2	6.33	2.53	122309	-11	3.24	4.63
paper2	187443	-8	6.53	4.13	183584	-14	3.60	4.17
paper3	118374	-19	5.98	4.01	116834	-10	3.16	2.85
paper4	37694	-6	4.32	0.00	37641	-7	1.66	4.22
paper5	35515	-10	4.03	3.47	35347	-6	1.58	1.27
paper6	92468	-15	5.91	3.21	91028	-14	2.95	2.71
pic	419631	0	7.96	7.41	412756	11	4.64	4.53
progc	95441	-8	6.04	3.31	93223	-3	3.00	3.00
progl	126468	-15	6.73	5.05	116035	-2	3.52	2.56
progp	86831	-7	6.43	5.60	82426	0	3.15	2.86
trans	145887	-16	6.98	4.87	132252	-33	3.70	2.97

TABLE 4.2. The effect of root weighting with the Zero-Redundancy estimator.

For Pearson's method (method_0) and Savoy's method (method_1) we use the permutation table as defined in [22]. This is also listed in Table 4.3. Below we list the code of Pearson's method.

```
byte hash_pearson(array C, integer n) :
  h[0] := 0;
  for i in 1..n loop
    h[i] := Tperm8[h[i-1] xor C[i]];
  end loop;
  return h[n];
end proc;

integer hash_method_0(array C, integer n) :
  c1 := C[1];
  h1 := hash_pearson(C, n);
  C[1] := (C[1] + 1) and 255;
  h2 = hash_pearson(C, n);
  C[1] := (C[1] + 1) and 255;
  h3 = hash_pearson(C, n);
  C[1] := c1;
  return (h1*2^16 + h2*2^8 + (h3 or 1)) and (TABLESIZE-1);
end proc;
```

```
Tperm8[256] = {
    1,  87,  49,  12, 176, 178, 102, 166, 121, 193,   6,  84, 249, 230,
   44, 163,  14, 197, 213, 181, 161,  85, 218,  80,  64, 239,  24, 226,
  236, 142,  38, 200, 110, 177, 104, 103, 141, 253, 255,  50,  77, 101,
   81,  18,  45,  96,  31, 222,  25, 107, 190,  70,  86, 237, 240,  34,
   72, 242,  20, 214, 244, 227, 149, 235,  97, 234,  57,  22,  60, 250,
   82, 175, 208,   5, 127, 199, 111,  62, 135, 248, 174, 169, 211,  58,
   66, 154, 106, 195, 245, 171,  17, 187, 182, 179,   0, 243, 132,  56,
  148,  75, 128, 133, 158, 100, 130, 126,  91,  13, 153, 246, 216, 219,
  119,  68, 223,  78,  83,  88, 201,  99, 122,  11,  92,  32, 136, 114,
   52,  10, 138,  30,  48, 183, 156,  35,  61,  26, 143,  74, 251,  94,
  129, 162,  63, 152,  170,  7, 115, 167, 241, 206,   3, 150,  55,  59,
  151, 220,  90,  53,  23, 131, 125, 173,  15, 238,  79,  95,  89,  16,
  105, 137, 225, 224, 217, 160,  37, 123, 118,  73,   2, 157,  46, 116,
    9, 145, 134, 228, 207, 212, 202, 215,  69, 229,  27, 188,  67, 124,
  168, 252,  42,   4,  29, 108,  21, 247,  19, 205,  39, 203, 233,  40,
  186, 147, 198, 192, 155,  33, 164, 191,  98, 204, 165, 180, 117,  76,
  140,  36, 210, 172,  41,  54, 159,   8, 185, 232, 113, 196, 231,  47,
  146, 120,  51,  65,  28, 144, 254, 221,  93, 189, 194, 139, 112,  43,
   71, 109, 184, 209 };
```

TABLE 4.3. Pearson's permutation table.

Savoy's method uses a multiplicative hashing function. The code is given here.

```
integer hash_savoy(array C, integer n) :
  h[0] := 0;
  for i in 1..n loop
    h[i] := (h[i-1]*137+Tperm8[C[i]]) and 255;
  end loop;
  return h[n];
end proc;

integer hash_method_1(array C, integer n) :
  c1 := C[1];
  h1 := hash_savoy(C, n);
  C[1] := (C[1] + 1) and 255;
  h2 = hash_savoy(C, n);
  C[1] := (C[1] + 1) and 255;
  h3 = hash_savoy(C, n);
  C[1] := c1;
  return (h1*2^16 + h2*2^8 + (h3 or 1)) and (TABLESIZE-1);
end proc;
```

For the CTW hashing function (method_2) we can precompute the result of the hashing function. The hash function is just the permutation `Tperm8` of the last symbol in the string and the complete method uses three consequtive table entries, so we have

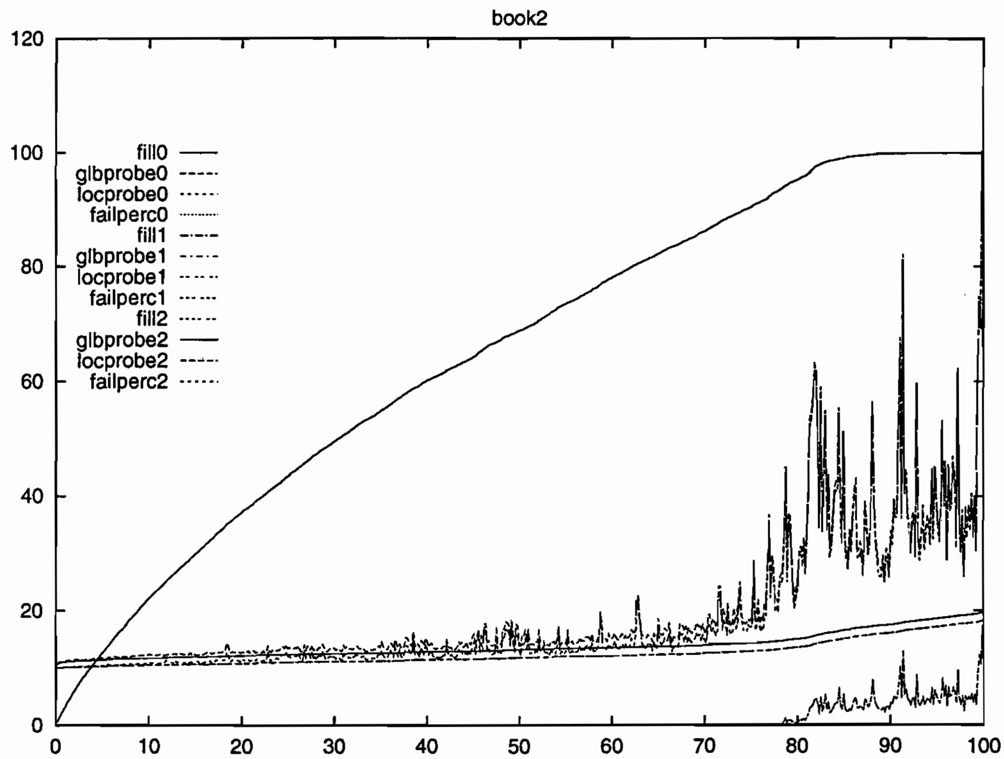


FIGURE 4.5. The performance of hashing for book2.

```
integer hash_method_2(array C, integer n) :
  cn := C[n];
  h1 := Tperm8[C[n]];
  C[n] := (C[n] + 1) and 255;
  h2 := Tperm8[C[n]];
  C[n] := (C[n] + 1) and 255;
  h3 := Tperm8[C[n]];
  C[n] := cn;
  return (h1*2^17 + h2*2^1 + h3*2 + 1) and (TABLESIZE-1);
end proc;
```

Precomputing this function results in the table listed in Table 4.4. Now using this table the hashing function becomes:

```
integer hash_method_2(array C, integer n) :
  return Tperm[C[n]] and (TABLESIZE-1);
end proc;
```

Note that the masking of the table entries with the hash table size (TABLESIZE) can also be incorporated in the table if only one hash table size is needed.

We selected four different files from the Calgary corpus to test the different hash functions. book2 is a large english text, geo is a file containing a numerical data set, obj2 is a computer object file, and paper1 is a short english text.

The graphs in Figures 4.5, 4.6, 4.7, and 4.8 display for each file as a function of the filelength (x-axis in percents of total length)

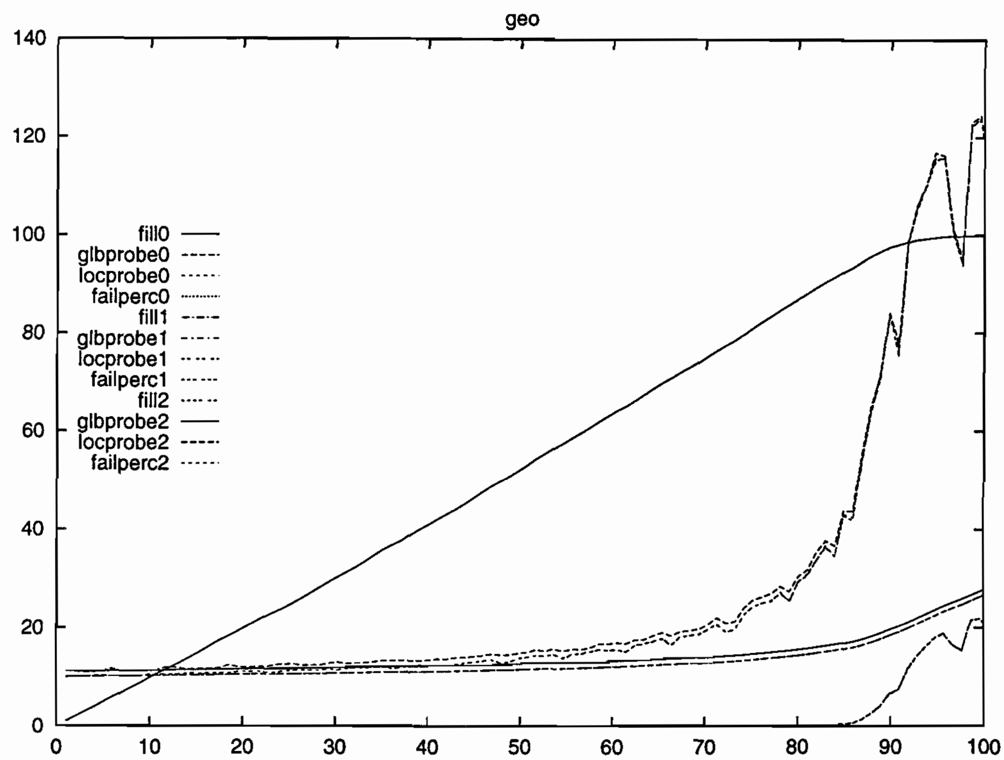


FIGURE 4.6. The performance of hashing for geo.

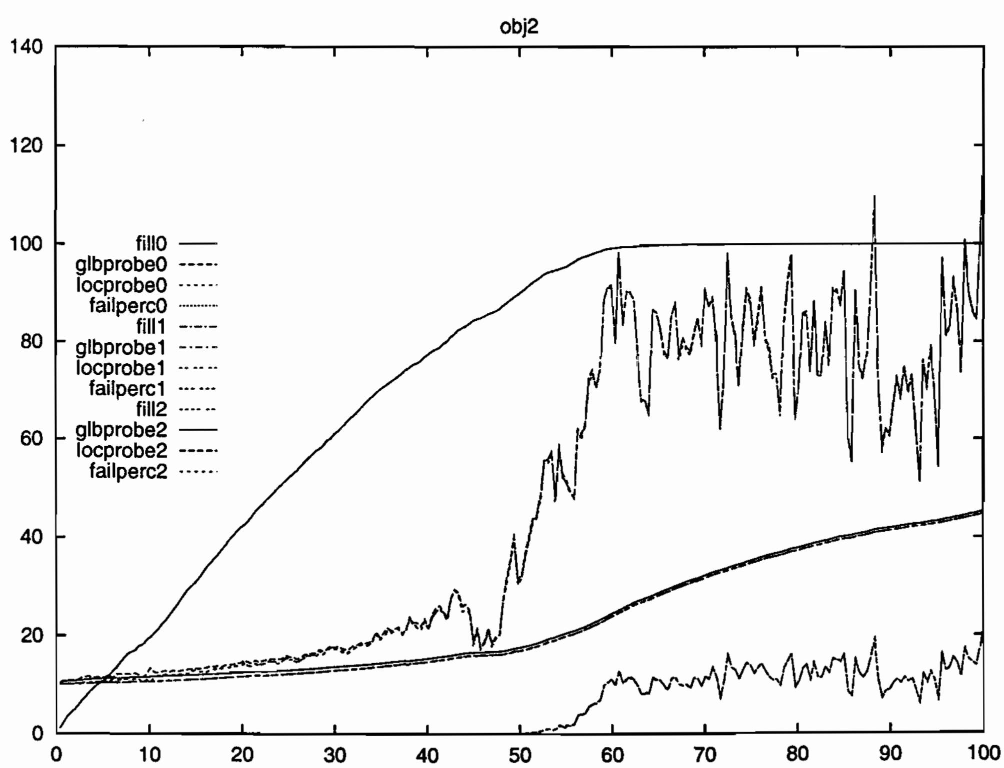


FIGURE 4.7. The performance of hashing for obj2.

```
integer Tperm[256] = {
  175715, 11428377,  6429025,  1663333, 23160013, 23383373, 13454579,
21820291, 15958541, 25300137,   829939, 11137997, 32754777, 30169415,
 5850653, 21372299,  1936299, 25930603, 28011331, 23806635, 21146549,
11252897, 28614785, 10519007,  8511025, 31338949,  3261913, 29743389,
31005773, 18632081,  5083357, 26271075, 14508753, 23253199, 13684507,
13573115, 18611199, 33291877, 33449115,  6593227, 10144419, 13279781,
10626139,  2382529,  5947455, 12599229,  4176947, 29110999,  3331965,
14122125, 24939693,  9219547, 11394017, 31187013, 31474833,  4493797,
 9561129, 31730093,  2731497, 28174791, 32098091, 29830103, 19650243,
30852053, 12833907, 30700077,  7482489,  2914805,  7992485, 32810335,
10837921, 23044107, 27265791,   720783, 16748255, 26140285, 14581007,
 8196081, 17822045, 32595283, 22893479, 22259317, 27686021,  7636277,
 8729813, 20239751, 13993963, 25684823, 32200227, 22422391,  2324333,
24604007, 23946753, 23462375,   124681, 31918193, 17330473,  7415959,
19437313,  9896203, 16845629, 17513673, 20760837, 13174013, 17104055,
16561691, 11934515,  1782765, 20180401, 32354743, 28423919, 28765833,
15632831,  9027229, 29269159, 10266289, 10924435, 11637447, 26396405,
13038615, 15996601,  1488961, 12075281,  4264165, 17884265, 14968853,
 6821141,  1381437, 18103393,  3957103,  6385465, 24066119, 20465275,
 4618805,  8008991,  3481237, 18781687,  9828029, 32947459, 12387141,
16991359, 21266225,  8335701, 20009999, 22286055,   976719, 15159267,
22012829, 31693831, 27002669,   470127, 19689079,  7239471,  7811001,
19904693, 28882027, 11823663,  6958855,  3081979, 17234779, 16472607,
22683613,  2088095, 31235775, 10403507, 12497441, 11673811,  2151187,
13833155, 18072513, 29606323, 29471553, 28524619, 20990711,  4912877,
16182419, 15503877,  9569595,   342621, 20602089,  6088723, 15209251,
 1254157, 19074505, 17680799, 29990825, 27240853, 27891119, 26586763,
28216267,  9161271, 30029689,  3635335, 24676089,  8845649, 16339449,
22149205, 33051657,  5507131,   539353,  3856427, 14167023,  2879015,
32384923,  2595407, 26890135,  5216211, 26726993, 30560629,  5338407,
24455053, 19369345, 26050871, 25245251, 20333385,  4409727, 21593797,
25085337, 12949835, 26823529, 21719275, 23653017, 15374617, 10033225,
18368933,  4826457, 27613267, 22565485,  5401919,  7159313, 20844915,
 1143761, 24367331, 30466953, 14911951, 25808479, 30301989,  6235377,
19198055, 15754883,  6718009,  8534305,  3744253, 19004859, 33405627,
29014907, 12286853, 24872215, 25499361, 18276439, 14702223,  5672667,
 9362289, 14381475, 24224259, 27394735 };
```

TABLE 4.4. The CTW hashing table.

- the hash table fill degree, in percents. The table size is choosen such that the table will become full for each file. The y-axis displays the fill degree in percents of the total table size.

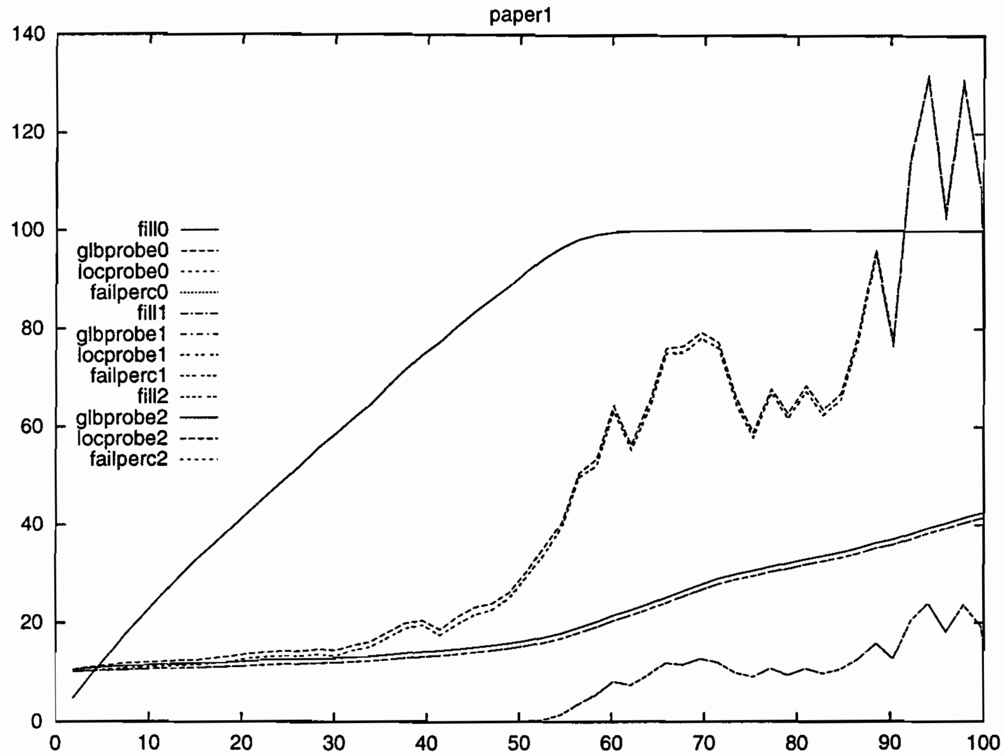


FIGURE 4.8. The performance of hashing for `paper1`.

- the global average number of probes in the table needed to find an entry, of fail after 50 probes. The y-axis displays the number of probes times 10.
- the local average number of probes in the last 1000 characters. The y-axis displays the number of probes times 10.
- the fraction of failed searches over the last 1000 characters. The y-axis displays the fraction in percents.

These four parameters are shown for each hashing function and are indexed as: Pearson's method (numbered 0), Savoy's method (numbered 1) and the CTW method (numbered 2).

We observe that the Pearson and Savoy methods are very similar and the CTW method requires about 10 percent more probes than the other two methods.

In order to compare the failure rate, which might influence the compression performance of the total algorithm, we include graphs that plot the failure rate of Pearson's method or Savoy's method and the failure rate difference between these methods and the CTW method for each of the four Calgary files. The results are shown in Figure 4.9.

From these figures we conclude that there is no real difference in failure rate between these three hashing functions.

Because the failure rates are almost identical and the CTW method needs about 10 % more probes than the other two but is much simpler and faster, we conclude that the CTW method is an appropriate choice.

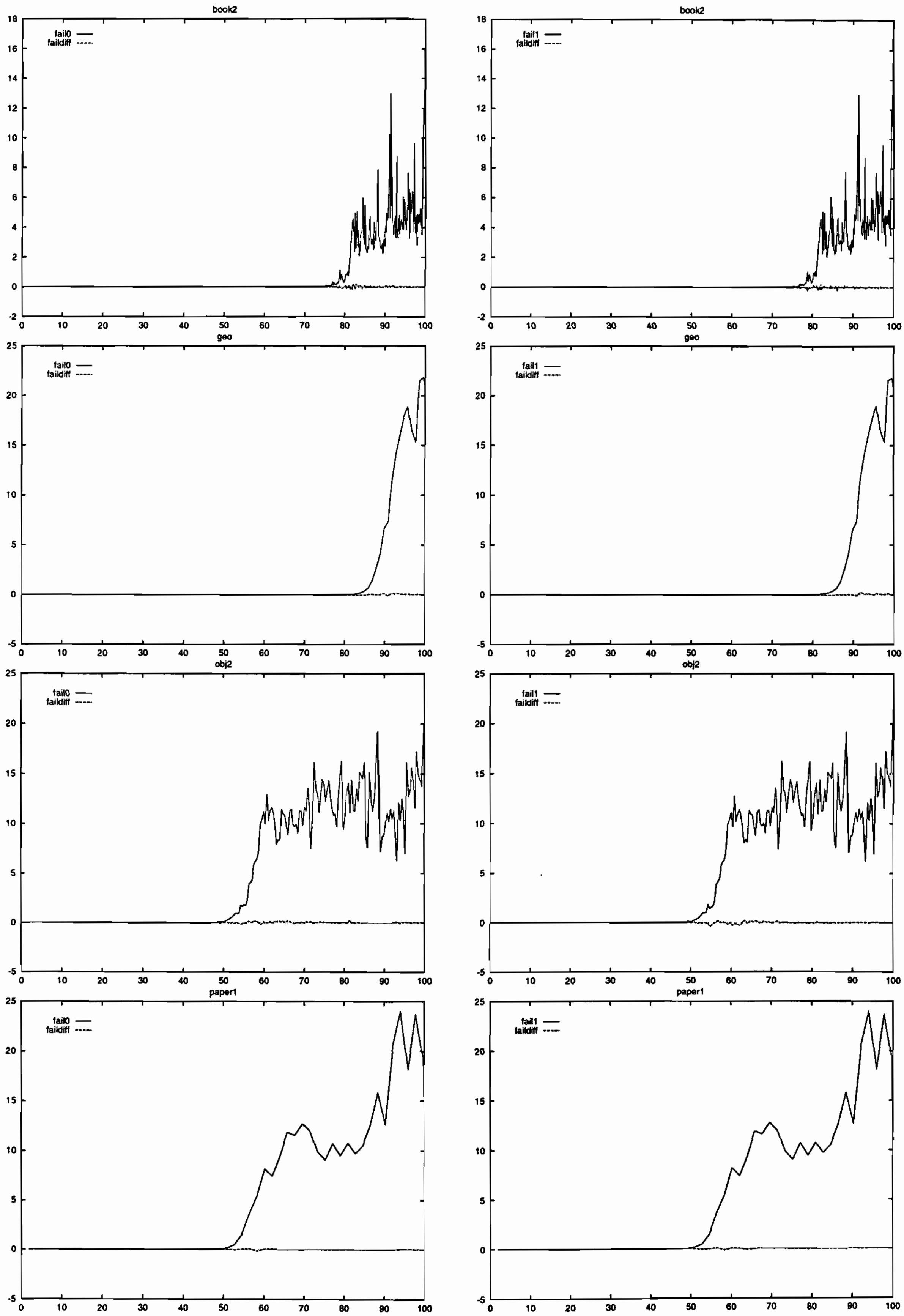


FIGURE 4.9. The failure rate

CHAPTER 5

Weighting Arithmetic

1. Implementation of Weighting in CTW-1

We have seen in the previous chapter that in CTW-1 we used the idea of "binary decomposition" to obtain eight context trees having binary estimators in all nodes, one tree for each bit in a byte (or ASCII symbol). A second idea was to "weight only at byte-boundaries". These two ideas together lead to a 256-ary context tree with binary estimators in each node. Moreover each 256-ary context extension was implemented as a binary rearch tree (see Figure 5.1).

Each node contains three pointers, one left- and one right-pointer to search in the current level and in addition to these two inter-level pointers a next-pointer leading to the next level (intra-level pointer).

Moreover a record contained an a- and a b-count, i.e. the number of zeroes and ones that occurred together with the context that corresponds to the record.

Finally the record contained four probabilities. The first (P_e) is the estimated probability corresponding to the node. The second probability is the weighted probability (P_w) of the node. Then there is a probability (P_m) corresponding to the missing counts (we will address this concept in section 7 of this chapter). Finally in each node there is a probability P_w^{tot} that is the product of the weighted probabilities of all nodes of the subtree starting in the node. This total weighted probability makes it possible to manipulate rather easily with products of "large" numbers of weighted probabilities.

For each pointer 4 bytes were needed. Also each count was allocated in 4 bytes. A probability consisted of a 2-byte mantissa and a 4-byte exponent. The CTW-information

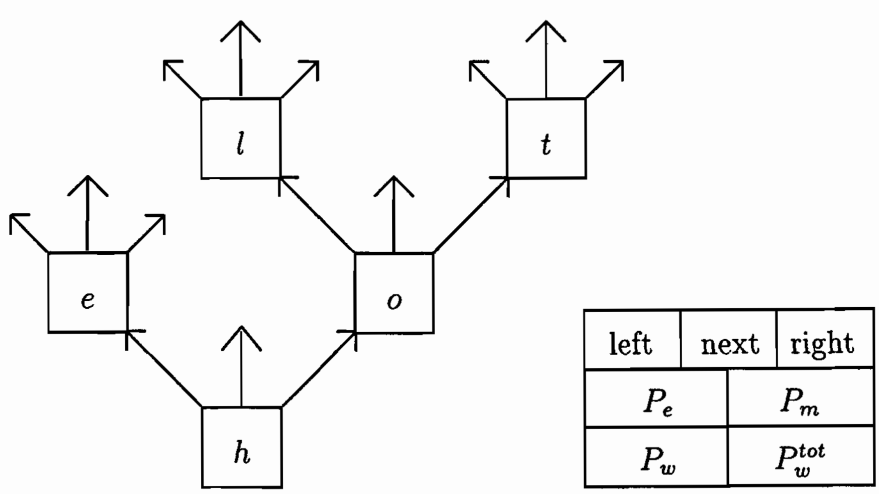


FIGURE 5.1. Binary search tree and record structure.

in each record consists of the counts together with the probabilities. Therefore each record in CTW-1 contained 32 bytes of CTW-information.

While in CTW-1 the probabilities were floating point numbers that corresponded to block probabilties, in CTW-2 we will use log-likelihoodratios of conditional probabilities. This change of view will give a considerable decrease of needed storage space as we shall see. Also the counts are stored more efficiently.

2. An idea: Consider Quotients of Probabilities

Let s be an internal node (not a leaf) in the context tree $\mathcal{T}_D$. Consider the quotient of the *conditional* weighted probabilities $P_w^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)$ and $P_w^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)$. For this quotient we obtain

$$\begin{aligned}
&\frac{P_w^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)}{P_w^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)} \\
&= \frac{P_w^s(x_1^{t-1}, X_t = 0|x_{1-D}^0)}{P_w^s(x_1^{t-1}, X_t = 1|x_{1-D}^0)} \\
&= \frac{P_e^s(x_1^{t-1}, X_t = 0|x_{1-D}^0) + P_w^{0s}(x_1^{t-1}, X_t = 0|x_{1-D}^0) P_w^{1s}(x_1^{t-1}, X_t = 0|x_{1-D}^0)}{P_e^s(x_1^{t-1}, X_t = 1|x_{1-D}^0) + P_w^{0s}(x_1^{t-1}, X_t = 1|x_{1-D}^0) P_w^{1s}(x_1^{t-1}, X_t = 1|x_{1-D}^0)} \\
&= \frac{P_e^s(x_1^{t-1}|\cdot) P_e^s(X_t = 0|x_1^{t-1}, \cdot) + P_w^{0s}(x_1^{t-1}|\cdot) P_w^{0s}(X_t = 0|x_1^{t-1}, \cdot) P_w^{1s}(x_1^{t-1}|\cdot)}{P_e^s(x_1^{t-1}|\cdot) P_e^s(X_t = 1|x_1^{t-1}, \cdot) + P_w^{0s}(x_1^{t-1}|\cdot) P_w^{0s}(X_t = 1|x_1^{t-1}, \cdot) P_w^{1s}(x_1^{t-1}|\cdot)} \\
&= \frac{\frac{P_e^s(x_1^{t-1}|x_{1-D}^0)}{P_w^{0s}(x_1^{t-1}|x_{1-D}^0) P_w^{1s}(x_1^{t-1}|x_{1-D}^0)} P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0) + P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0)}{\frac{P_e^s(x_1^{t-1}|x_{1-D}^0)}{P_w^{0s}(x_1^{t-1}|x_{1-D}^0) P_w^{1s}(x_1^{t-1}|x_{1-D}^0)} P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0) + P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0)}. \qquad (63)
\end{aligned}$$

Here we use in the second equality the main CTW-definition (see (12) in chapter 3). In the third equality we have split out all probabilities in a block-part depending on x_1^{t-1} and a conditional part for $X_t = 0$ or 1 given x_1^{t-1}. The condition x_{1-D}^0 is denote by a $\cdot$ to save space. We also assumed that $0s$ and not $1s$ is a suffix of the context x_1^{t-1}, x_{1-D}^0 of x_t. In the fourth equality we have divided both nominator and enumerator by the product $P_w^{0s}(x_1^{t-1}|x_{1-D}^0) P_w^{1s}(x_1^{t-1}|x_{1-D}^0)$.

Next define

$$\eta^s(x_1^{t-1}|x_{1-D}^0) \triangleq \frac{P_w^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)}{P_w^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)} \qquad (64)$$

and observe that the conditional weighted probabilities $P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0)$ and $P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0)$ can be determined as follows:

$$\begin{aligned}
P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0) &= \frac{\eta^{0s}(x_1^{t-1}|x_{1-D}^0)}{1 + \eta^{0s}(x_1^{t-1}|x_{1-D}^0)} \text{ and} \\
P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0) &= \frac{1}{1 + \eta^{0s}(x_1^{t-1}|x_{1-D}^0)}. \qquad (65)
\end{aligned}$$

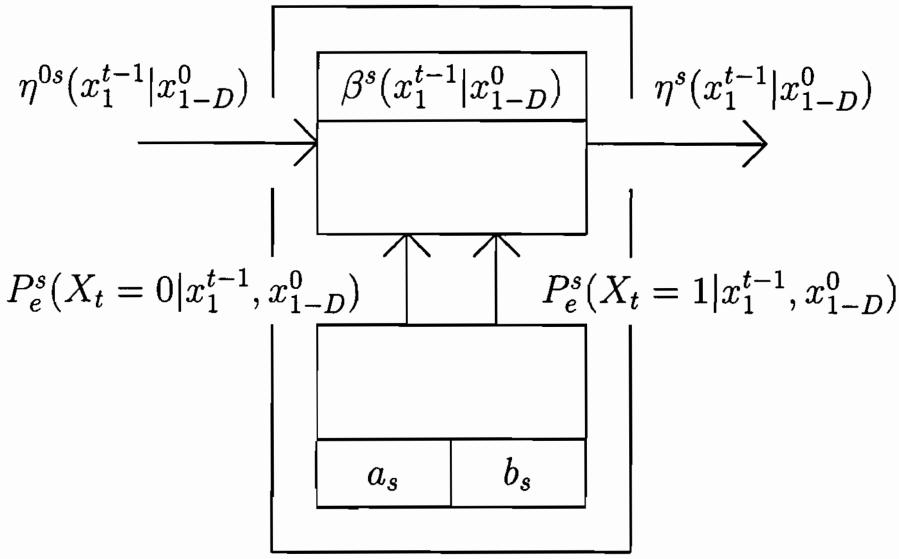


FIGURE 5.2. Variables contained in and information flow through node s.

If we furthermore define

$$\beta^s(x_1^{t-1}|x_{1-D}^0) \triangleq \frac{P_e^s(x_1^{t-1}|x_{1-D}^0)}{P_w^{0s}(x_1^{t-1}|x_{1-D}^0)P_w^{1s}(x_1^{t-1}|x_{1-D}^0)}, \tag{66}$$

we can rewrite (63) in the following way:

$$\eta^s(x_1^{t-1}|x_{1-D}^0) = \frac{\beta^s(x_1^{t-1}|x_{1-D}^0)P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0) + P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0)}{\beta^s(x_1^{t-1}|x_{1-D}^0)P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0) + P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0)}, \tag{67}$$

where both $P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0)$ and $P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0)$ can be determined from $\eta^{0s}(x_1^{t-1}|x_{1-D}^0)$.

If we assume (see Figure 5.2) that in node s the counts $a_s(x_1^{t-1}|x_{1-D}^0)$ and $b_s(x_1^{t-1}|x_{1-D}^0)$ are stored, as well as the quotient $\beta^s(x_1^{t-1}|x_{1-D}^0)$, we obtain the following sequence of operations:

- The quotient $\eta^{0s}(x_1^{t-1}|x_{1-D}^0)$ enters node s. We assume that node $0s$ emitted this quotient[1].
- Inside the node s, the conditional weighted probabilities $P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0)$ and $P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0)$ are determined from the incoming $\eta^{0s}(x_1^{t-1}|x_{1-D}^0)$ as in (65).
- The conditional estimated probabilities are determined from the counts $a_s(x_1^{t-1}|x_{1-D}^0)$ and $b_s(x_1^{t-1}|x_{1-D}^0)$ as suggested by Krichevsky and Trofimov [**15**], i.e.:

$$\begin{aligned} P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0) &= \frac{a_s(x_1^{t-1}|x_{1-D}^0) + 1/2}{a_s(x_1^{t-1}|x_{1-D}^0) + b_s(x_1^{t-1}|x_{1-D}^0) + 1} \\ P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0) &= \frac{b_s(x_1^{t-1}|x_{1-D}^0) + 1/2}{a_s(x_1^{t-1}|x_{1-D}^0) + b_s(x_1^{t-1}|x_{1-D}^0) + 1} \end{aligned} \tag{68}$$

- Now the outgoing quotient $\eta^s(x_1^{t-1}|x_{1-D}^0)$ can be computed using (67).

[1] When the context x_1^{t-1}, x_{1-D}^0 passes through $1s$ the node $1s$ delivers an incoming quotient.

- The quotient $\beta^s(\cdot)$ is now updated with the new x_t. This is done as described below:

$$\beta^s(x_1^{t-1}, x_t|x_{1-D}^0) = \begin{cases} \beta^s(x_1^{t-1}|x_{1-D}^0) \cdot \frac{P_e^s(X_t=0|x_1^{t-1},x_{1-D}^0)}{P_w^{0s}(X_t=0|x_1^{t-1},x_{1-D}^0)} & \text{if } x_t = 0, \\ \beta^s(x_1^{t-1}|x_{1-D}^0) \cdot \frac{P_e^s(X_t=1|x_1^{t-1},x_{1-D}^0)}{P_w^{0s}(X_t=1|x_1^{t-1},x_{1-D}^0)} & \text{if } x_t = 1. \end{cases} \tag{69}$$

- Finaly the counts $a_s(x_1^{t-1}|x_{1-D}^0)$ and $b_s(x_1^{t-1}|x_{1-D}^0)$ are updated. Again x_t determines how.

$$\begin{aligned} &(a_s(x_1^{t-1}, x_t|x_{1-D}^0), b_s(x_1^{t-1}, x_t|x_{1-D}^0)) \\ &= \begin{cases} (a_s(x_1^{t-1}, x_t|x_{1-D}^0) + 1, b_s(x_1^{t-1}, x_t|x_{1-D}^0)) & \text{if } x_t = 0, \\ (a_s(x_1^{t-1}, x_t|x_{1-D}^0), b_s(x_1^{t-1}, x_t|x_{1-D}^0) + 1) & \text{if } x_t = 1. \end{cases} \end{aligned} \tag{70}$$

We see that inside the node s there is a *switch* that controls the mixture between the incoming (external) quotient $\eta^{0s}(x_1^{t-1}|x_{1-D}^0) = P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0)/P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0)$ and the (internal) quotient $P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)/P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)$. The mixture is determined by the quotient $\beta^s(x_1^{t-1}|x_{1-D}^0)$. If s is a leaf in $\mathcal{T}_D$ then the outgoing $\beta^s(x_1^{t-1}|x_{1-D}^0)$ is simply $P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)/P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)$.

3. Logarithmic Representations

To make multiplying and dividing easy we represent all η's and β's by their logarithm[2]. We assume that these logarithms are represented by fixed point numbers with a, say, m-bit fractional part. The size of the integer part is to be discussed later.

This logarithmic representation leads to several problems. The first problem is that apart from multiplying and dividing we have to add two quantities regularly. To do this we use a table containing the Jacobian logarithm $\log(1 + 2^x)$ for $x \leq 0$. Now if $p' \geq p''$ then

$$\begin{aligned} \log(p' + p'') &= \log p'(1 + \frac{p''}{p'}) \\ &= \log p' + \log(1 + \frac{p''}{p'}) \\ &= \log p' + \log(1 + 2^{(\log p'' - \log p')}), \end{aligned} \tag{71}$$

with $\log p'' - \log p' \leq 0$. The number of entries in this table is limited as we shall see later.

A second problem is that we need to compute the logarithms of the estimated probabilities $P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)$ and $P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)$, i.e.

$$\begin{aligned} &\log P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0) \\ &\quad = \log(2a_s(x_1^{t-1}|x_{1-D}^0) + 1) - 1 - \log(a_s(x_1^{t-1}|x_{1-D}^0) + b_s(x_1^{t-1}|x_{1-D}^0) + 1) \\ &\log P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0) \\ &\quad = \log(2b_s(x_1^{t-1}|x_{1-D}^0) + 1) - 1 - \log(a_s(x_1^{t-1}|x_{1-D}^0) + b_s(x_1^{t-1}|x_{1-D}^0) + 1). \end{aligned} \tag{72}$$

To be able to do this we use log-tables. To keep the number of entries in these log-tables finite however, we have to bound the counts. Therefore we assume that we count the zeros and ones in registers that are, say, k bit wide. Now suppose that we reach the point where

[2] Remember that the base of the log is 2.

$a := 2^k$ (while $b = b^* < 2^k$). Since the registers can not contain this value of a, both a and b are divided by 2, i.e. $a := 2^{k-1}$ and $b := \lceil b^*/2 \rceil$, where $\lceil g \rceil$ ($\lfloor g \rfloor$) denotes the smallest (largest) integer not smaller (larger) than g. This leads to a redundancy increase of course. This is investigated in appendix 8.

Since now both $0 \leq a_s(\cdot) \leq 2^k - 1$ and $0 \leq b_s(\cdot) \leq 2^k - 1$ the arguments of the logarithms in (72) are bounded and we can form a table.

4. Standard Bit Allocations

4.1. Counts. In the standard configuration we assume that $k = 8$, i.e. both counts are stored in a byte, and $a_s(\cdot)$ and $b_s(\cdot)$ are both upperbounded by $2^k - 1 = 255$.

4.2. The log-table $L[\cdot]$. For the log-table this implies that the largest required entry in the log-table is $2 \cdot 255 + 1 = 511$. Suppose that we form a table with entries $256, 257, \cdots, 511$. We can now find all logarithms of $g = 1, 2, \cdots, 255$ in this table by noting that $\log g = \log(g \cdot 2^h) - h$ for integer h such that $256 \leq g \cdot 2^h \leq 511$. This integer h can always be found. In other words, a table with 2^k entries suffices.

What about the accuracy m of the table elements, i.e. the number of fractional bits in which the logarithms are specified? To see how large m should be, consider the derivative of $\eta^s(x_1^{t-1}|x_{1-D}^0)$ with respect to $p := P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)$, i.e.

$$\begin{aligned} \frac{d \log \frac{p}{1-p}}{dp} &= \frac{1}{\ln 2}\left(\frac{1}{p} + \frac{1}{1-p}\right) \\ &= \frac{1}{\ln 2}\frac{1}{p(1-p)} \\ &\geq \frac{4}{\ln 2} = 5.7708. \end{aligned} \tag{73}$$

The minimum is achieved for $p = 1 - p = 1/2$. We are now interested in the smallest possible change in $\eta^s(x_1^{t-1}|x_{1-D}^0)$ caused by a change in a or b. The smallest change in p is roughly $\frac{1}{2\cdot(2^k-1)+1} \approx 2^{-k-1} = \frac{1}{512}$. Therefore the smallest change in $\eta^s(x_1^{t-1}|x_{1-D}^0)$ is $\frac{4\cdot 2^{-k-1}}{\ln 2} = \frac{2^{-k+1}}{\ln 2} = 0.0113$. To be able to represent this smallest change in $\eta^s(x_1^{t-1}|x_{1-D}^0)$ we need at least $m = k - 1 = 7$ bits. Therefore we form the log-table $L[\cdot]$ as follows:

$$L[j] := \lfloor 2^m \cdot \log(j) + 1/2 \rfloor \text{ for } j = 2^k, 2^k + 1, \cdots, 2^{k+1} - 1, \tag{74}$$

with $m = k - 1$. Note that the since $k \cdot 2^m \leq L[\cdot] \leq (k+1) \cdot 2^m$ we need only $m + 1$ bits to store each table element.

4.3. The Jacobian table $J[\cdot]$. How about the table containing the Jacobian logarithm? It is obvious that the accuracy of the table elements should be m again, i.e. table elements are fixed point numbers having a fractional part of m bits wide. Moreover the entries are also logarithms, i.e. fixed point numbers with an m bit fractional part. Therefore we define the Jacobian table as

$$J[i] := \lfloor 2^m \cdot \log(1 + 2^{i/2^m}) + 1/2 \rfloor \text{ for } i = \cdots, -2, -1, 0. \tag{75}$$

Values of $y = i \cdot 2^m$ such that $\log(1+2^y) < 2^{-m-1}$ will lead to table elements equal to zero and are not informative. Therefore for

$$y < \log(2^{2^{-m-1}} - 1) \approx \log(2^{-m-1} \cdot \ln 2) = \log \ln 2 - m - 1 = -m - 1.5288, \tag{76}$$

we need no table elements. In the standard case $m = k - 1 = 7$, hence we need elements only for $-k - 0.5288 \leq y \leq 0$ or, say, for $-9 \leq y \leq 0$. Hence $i = -9 \cdot 128, \cdots, -2, -1, 0$. The number of entries in the standard setting is roughly $(k+1)2^{k-1} = 1152$. Note that since $0 \leq J[\cdot] \leq 2^m$ we need only $m + 1$ bits to store the table elements.

4.4. The range of $\eta^s(\cdot)$. Since $0 \leq a_s(\cdot) \leq 2^k - 1$ and $0 \leq b_s(\cdot) \leq 2^k - 1$ we can write for the quotient

$$\frac{1/2}{2^k - 1 + 1/2} \leq \eta^s(x_1^{t-1}|x_{1-D}^0) \leq \frac{2^k - 1 + 1/2}{1/2}, \tag{77}$$

or roughly $2^{-k-1} \leq \eta^s(x_1^{t-1}|x_{1-D}^0) \leq 2^{k+1}$ or $-k-1 \leq \log \eta^s(x_1^{t-1}|x_{1-D}^0) \leq k+1$. Therefore, apart from the m fractional bits, $\lceil \log 2(k+1) \rceil$ bits are needed for the integer part of $\eta^s(\cdot)$. In the standard setting ($k = 8$) this leads to 5 bits for the integer part.

4.5. Bounding the range of $\beta^s(\cdot)$. Since $\beta^s(\cdot)$ has to be stored in node s we must bound its range. To do this we can argue as follows. We want the β to have a range that is large enough to switch to either the (external) quotient $P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0)/P_w^{0s}(X_t = 1|x_1^{t-1}, x_{1-D}^0)$ or the (internal) quotient $P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)/P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)$, no matter how big the difference between these quotients is. Suppose e.g. that the external quotient is small and the internal one is large. Then $\log P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0) \leq \log P_w^{0s}(X_t = 0|x_1^{t-1}, x_{1-D}^0) + (k+1)$ by (77). Note that by the definition of the Jacobian table, the sum of two terms that differ more in log than $m+2$ is completely determined by the larger term. The smaller term just does not matter. Keeping this in mind it is useless to make $\log \beta$ smaller than $-m - k - 3$. Similarly we can argue that it is useless to make $\log \beta$ larger than $m + k + 3$. The range of $\beta^s(\cdot)$ would then become

$$2^{-m-k-3} \leq \beta^s(x_1^{t-1}|x_{1-D}^0) \leq 2^{m+k+3}, \tag{78}$$

or $-2(k+1) \leq \log \beta^s(x_1^{t-1}|x_{1-D}^0) \leq 2(k+1)$ if we substitute $m = k - 1$. For $k = 8$ this bound yields 18 in absolute value. Therefore we need, besides the $m = 7$ fractional bits, $\lceil \log 4(k+1) \rceil$ bits more for the integer part of $\beta^s(\cdot)$. In the standard setting ($k = 8$) 6 bits are required for the integer part of β.

To achieve the real CTW behaviour β, should not be bounded at all. However bounding β decreases the storage space needed for β and also improves the performance quite often because of non-stationary parts in the data. In a system with bounded β it takes less time to forget the past. A similar effect results from scaling the counts a_s and b_s.

4.6. Storage space in each node. The total amount of storage that is needed for one record is now two times 8 bit for the counts plus 7 fractional and 6 integer part bits for β, which brings us to 29 bits. This is a lot less than what was used in CTW-1 (32 bytes).

5. Zero-Redundancy Estimator

The codelength for a sequence containing T zeros or ones, if we use the Krichevsky-Trofimov estimator (see (10) is roughly equal to $\frac{1}{2}\log T + 1$ bit. This is quite large if we realize that such a sequence does not contain any information. Therefore instead of the Krichevsky-Trofimov estimator, we can use the so-called "zero-redundancy" estimator. This estimator is defined as

$$P_e^{zr}(a,b) \triangleq \begin{cases} \frac{1}{2}P_e(a,b) & \text{for } a>0, b>0, \\ \frac{1}{4}+\ \frac{1}{2}P_e(a,0) & \text{for } a>0, b=0, \\ \frac{1}{2}P_e(0,b) + \frac{1}{4} & \text{for } a=0, b>0, \text{ and} \\ 1 & \text{for } a=b=0, \end{cases} \tag{79}$$

where $P_e(a,b)$ is as defined in (8), i.e. the Krichevsky-Trofimov estimator.

In the case where the source sequence contains only zeroes ($a=T$ and $b=0$) the (parameter-) redundancy can be upper bounded as follows :

$$\log \frac{(1-\theta)^a \theta^b}{P_e^{zr}(a,b)} \le \log \frac{1}{\frac{1}{4}} = 2, \tag{80}$$

no matter how large T is. So we loose 2 bits and that's it. The same holds for a sequence that contains only ones.

When the sequence contains both zeroes and ones we get an increased redundancy

$$\log \frac{(1-\theta)^a \theta^b}{P_e^{zr}(a,b)} \le \log \frac{(1-\theta)^a \theta^b}{\frac{1}{2}P_e(a,b)} \le \frac{1}{2}\log T + 2, \tag{81}$$

so we loose one bit relative to the standard Krichevsky-Trofimov estimator (see 11).

Since in CTW-2 we need instead of block-probabilities conditional probabilities, we consider e.g. the conditional zero-redundancy estimated probability of a one given $a \ge 1$ zeros. Using (79) this probability can be rewritten as

$$\begin{aligned} P_e^{zr}(1|0^a) &= \frac{\frac{1}{2}P_e(a,1)}{\frac{1}{2}P_e(a,0)+\frac{1}{4}} \\ &= \frac{\frac{1}{2}P_e(a,0)\frac{\frac{1}{2}}{a+1}}{\frac{1}{2}P_e(a,0)+\frac{1}{4}} \\ &= \frac{P_e(a,0)}{2(a+1)P_e(a,0)+(a+1)}. \end{aligned} \tag{82}$$

Similarly

$$P_e^{zr}(0|0^a) = \frac{(2a+1)P_e(a,0)+(a+1)}{2(a+1)P_e(a,0)+(a+1)}. \tag{83}$$

If we furthermore note that $P_e(a+1,0) = P_e(a,0)\frac{a+\frac{1}{2}}{a+1}$, these two conditional probabilities can easily be calculated for $a = 1, 2, \cdots$. For $a=0$ we simply have that $P_e^{zr}(0) = P_e^{zr}(1) = \frac{1}{2}$.

Since a (and b) is bounded by 2^k-1 we can tabulate the logarithm of these probabilities. The accuracy is again m binary digits. More precisely

$$\begin{aligned}
M_-[a] &= \lfloor 2^m \cdot \log P_e^{zr}(1|0^a) + 1/2 \rfloor \\
M_+[a] &= \lfloor 2^m \cdot \log P_e^{zr}(0|0^a) + 1/2 \rfloor \text{ for } a = 1, 2, \cdots, 2^k - 1, \\
M_-[0] &= -2^m, \text{ and} \\
M_+[0] &= -2^m.
\end{aligned} \tag{84}$$

If we now encounter counts for which either a or b is zero, we use these two tables to find the log of the smallest and largest conditional zero-redundancy estimated probability. If both a and b are positive, we use the log-table $L[\cdot]$ to form the log of both probabilities as described in (72).

There is now one thing left to discuss about this concept. Note that in our argument to find the necessary range of β we assumed that $-(k+1) \leq \log \eta \leq (k+1)$. Note that this does not hold anymore for the zero redundancy estimator. Using the lower bound for the Krickevsky-Trofimov estimator (10), we can show that

$$\begin{aligned}
\eta^{zr} &= \frac{(2a+1)P_e(a,0) + (a+1)}{P_e(a,0)} \\
&= (2a+1) + \frac{a+1}{P_e(a,0)} \\
&\leq (2a+1) + 2(a+1)\sqrt{a} \\
&\leq 2^{k+1} + 2^{3k/2+1}.
\end{aligned} \tag{85}$$

Note that the difference with (77) is not very large and can be neglected.

6. Non-Binary Contexts

Although so far we have assumed that context symbols are binary, we should realize that in our software we are interested only in 256-ary context symbols[3]. This implies that the basic weighting formula (12) should be modified as follows:

$$P_w^s \triangleq \begin{cases} \frac{1}{2}P_e(a_s,b_s) + \frac{1}{2}\prod_{c=1,C} P_w^{cs} & \text{for } 0 \leq l(s) < D, \\ P_e(a_s,b_s) & \text{for } l(s) = D. \end{cases} \tag{86}$$

Here the context alphabet is assumed to be $\{1, 2, \cdots, C\}$. This new definition has minor consequences for the approach that we pursue in this chapter. It turns out that the only thing that changes is the definition of β. For the non-binary case the definition of β is

$$\beta^s(x_1^{t-1}|x_{1-D}^0) \triangleq \frac{P_e^s(x_1^{t-1}|x_{1-D}^0)}{\prod_{c=1,C} P_w^{cs}(x_1^{t-1}|x_{1-D}^0)}. \tag{87}$$

[3]The symbols that are to be coded are binary however. This setup is a result of the "weighting only at byte-boundaries"-idea and the "binary-decomposition". Both concepts are applied already in CTW-1.

The outgoing η^s follows from the incoming η^{cs} as described by (67) so nothing actually changes here. We also update β just like before, i.e.

$$\beta^s(x_1^{t-1}, x_t | x_{1-D}^0) = \begin{cases} \beta^s(x_1^{t-1}|x_{1-D}^0) \cdot \frac{P_e^s(X_t=0|x_1^{t-1},x_{1-D}^0)}{P_w^{cs}(X_t=0|x_1^{t-1},x_{1-D}^0)} & \text{if } x_t = 0, \\ \beta^s(x_1^{t-1}|x_{1-D}^0) \cdot \frac{P_e^s(X_t=1|x_1^{t-1},x_{1-D}^0)}{P_w^{cs}(X_t=1|x_1^{t-1},x_{1-D}^0)} & \text{if } x_t = 1, \end{cases} \tag{88}$$

if cs is a suffix of the context x_1^{t-1}, x_{1-D}^0.

7. Missing Contexts and Counts

In CTW-1 the eight context trees were stored as a dynamical structure, with records containing pointers to other records. When all storage space was reserved, we could not allocate any more records. In CTW-2 we use hashing to allocate the records that correspond to the treenodes. The more storage space is used, the higher the probability of not being able to allocate a record becomes. These two examples show that it is very well possible that we encounter the case where cs is a suffix of the context x_1^{t-1}, x_{1-D}^0 and where there is no node cs (or this node can not be created) although node s exists. This leads to a problem. We see that it is very easy in this case to maintain the block estimator $P_e(a_s, b_s)$ although it is impossible to maintain the product of the weighted probabilities of the descendants of s which is $\prod_{c=1,C} P_w^{cs}$ since some of these descendant nodes do not exist. Note that both probabilities are part of $\beta^s(\cdot)$ and are needed therefore.

In CTW-1 the counts that are missing are treated in a very simple way. For each missing count we multiply the P_w-product by 1/2. Here in CTW-2 we suggest another approach. Suppose that cs is a missing context. When this context occurs, P_e is updated, i,e. multiplied by some conditional probability, say z. To make life easy we also multiply the P_w-product by this number z. The first effect of this operation is that the outgoing $\eta^s(\cdot)$ from node s is simply the internal quotient $P_e^s(X_t = 0|x_1^{t-1}, x_{1-D}^0)/P_e^s(X_t = 1|x_1^{t-1}, x_{1-D}^0)$ when cs occurs. The second effect is that $\beta^s(\cdot)$ does not change since it is both multiplied with and divided by z. This solution is very simple to implement and it is very likely that z is a better estimate of the conditional probability of the missing symbol than 1/2.

8. Appendix: A Scaled Dirichlet Estimator

(This section is almost identical to a part of [51].)

For memoryless sources with unknown parameter θ (the probability of generating a one), it is reasonable to assign the block probability $P_c(x_1 \cdots x_T) = P_e(a, b)$ to a sequence $x_1 \cdots x_T$ containing a zeros and b ones where

$$P_e(a,b) \triangleq \frac{\frac{1}{2} \cdot \frac{3}{2} \cdot \ldots \cdot (a - \frac{1}{2}) \cdot \frac{1}{2} \cdot \frac{3}{2} \cdot \ldots \cdot (b - \frac{1}{2})}{1 \cdot 2 \cdot \ldots \cdot (a+b)} \text{ for } a > 0 \text{ and } b > 0, \text{ etc.} \tag{89}$$

This distribution, which allows sequential updating and therefore arithmetic coding, was suggested by Krichevsky and Trofimov **[15]** and is referred to as Dirichlet estimator. It guarantees uniform convergence of the *parameter redundancy*, i.e. for any sequence $x_1 \cdots x_T$ with actual probability $P_a(x_1^T) = (1-\theta)^a \theta^b$, it can be shown that

$$\log \frac{P_a(x_1^T)}{P_c(x_1^T)} \leq \frac{1}{2} \log T + 1 \text{ for all } \theta \in [0, 1]. \tag{90}$$

We assume that the base of the log is 2. The bound (90) follows from the lemma below (for a proof see appendix 10 in chapter 3 taken from [**57**]).

LEMMA 8.1. *For* $a+b \geq 1$

$$\frac{1}{2} \cdot \frac{1}{\sqrt{a+b}}\left(\frac{a}{a+b}\right)^a\left(\frac{b}{a+b}\right)^b \leq P_e(a,b) \leq \sqrt{\frac{2}{\pi}} \cdot \frac{1}{\sqrt{a+b}}\left(\frac{a}{a+b}\right)^a\left(\frac{b}{a+b}\right)^b. \tag{91}$$

8.1. First rescaling operation. Next we assume that we count the zeros and ones in registers that are k bit wide. Suppose that we reach the point where $a := 2^k$ (while $b = b^* < 2^k$). Since the registers can not contain this value of a, both a and b are divided by 2, i.e. $a := 2^{k-1}$ and $b := \lceil b^*/2 \rceil$, where $\lceil g \rceil$ ($\lfloor g \rfloor$) denotes the smallest (largest) integer not smaller (larger) than g. Now we continue with these values of the counts as before and the resulting block estimator $\hat{P}_e(a,b)$ can be decomposed as

$$\hat{P}_e(a,b) = \frac{P_e(2^k,b^*)P_e(a-2^{k-1}, b-\lfloor b^*/2 \rfloor)}{P_e(2^{k-1}, \lceil b^*/2 \rceil)}, \text{ for } a \geq 2^k, b \geq b^*. \tag{92}$$

We are interested now in the *scaling redundancy*, i.e. the increase in codewordlength resulting from this rescaling operation. Therefore we bound $\hat{P}_e(a,b)/P_e(a,b)$ from below.

First assume that b^* is **even**, then $\lceil b^*/2 \rceil = \lfloor b^*/2 \rfloor = b^*/2$. From (92) and lemma 8.1 we obtain

$$\begin{aligned}\frac{\hat{P}_e(a,b)}{P_e(a,b)} \geq & \frac{\pi}{8} \cdot \sqrt{\frac{(2^{k-1}+b^*/2)(a+b)}{(2^k+b^*)(a-2^{k-1}+b-b^*/2)}} \cdot \\ & \frac{\left(\frac{2^k}{2^k+b^*}\right)^{2^k}\left(\frac{b^*}{2^k+b^*}\right)^{b^*}\left(\frac{a-2^{k-1}}{a-2^{k-1}+b-b^*/2}\right)^{a-2^{k-1}}\left(\frac{b-b^*/2}{a-2^{k-1}+b-b^*/2}\right)^{b-b^*/2}}{\left(\frac{2^{k-1}}{2^{k-1}+b^*/2}\right)^{2^{k-1}}\left(\frac{b^*/2}{2^{k-1}+b^*/2}\right)^{b^*/2}\left(\frac{a}{a+b}\right)^a\left(\frac{b}{a+b}\right)^b}.\end{aligned} \tag{93}$$

The squareroot-factor can be lowerbounded by $1/\sqrt{2}$. Next observe that

$$\frac{\left(\frac{2^k}{2^k+b^*}\right)^{2^k}\left(\frac{b^*}{2^k+b^*}\right)^{b^*}}{\left(\frac{2^{k-1}}{2^{k-1}+b^*/2}\right)^{2^{k-1}}\left(\frac{b^*/2}{2^{k-1}+b^*/2}\right)^{b^*/2}} = \left(\frac{2^{k-1}}{2^{k-1}+b^*/2}\right)^{2^{k-1}}\left(\frac{b^*/2}{2^{k-1}+b^*/2}\right)^{b^*/2}. \tag{94}$$

From the log-sum inequality (see Csiszár and Körner [**5**]) we obtain that

$$\begin{aligned}\left(\frac{2^{k-1}}{2^{k-1}+b^*/2}\right)^{2^{k-1}}\left(\frac{a-2^{k-1}}{a-2^{k-1}+b-b^*/2}\right)^{a-2^{k-1}} &\geq \left(\frac{a}{a+b}\right)^a, \text{ and} \\ \left(\frac{b^*/2}{2^{k-1}+b^*/2}\right)^{b^*/2}\left(\frac{b-b^*/2}{a-2^{k-1}+b-b^*/2}\right)^{b-b^*/2} &\geq \left(\frac{b}{a+b}\right)^b.\end{aligned} \tag{95}$$

From all this we conclude that for even values of b^*

$$\hat{P}_e(a,b) \geq \frac{\pi}{8\sqrt{2}} \cdot P_e(a,b). \tag{96}$$

For **odd** values of b^* the situation is slightly more complicated. First observe that $\lceil b^*/2 \rceil = \frac{b^*+1}{2}$ and $\lfloor b^*/2 \rfloor = \frac{b^*-1}{2}$ and rewrite (92) as

$$\frac{\hat{P}_e(a,b)}{P_e(a,b)} = \frac{P_e(2^k, b^*+1)P_e(a-2^{k-1}, b-\frac{b^*-1}{2})}{P_e(2^{k-1}, \frac{b^*+1}{2})P_e(a,b+1)} \cdot \frac{\frac{b+\frac{1}{2}}{a+b+1}}{\frac{b^*+\frac{1}{2}}{2^k+b^*+1}}. \tag{97}$$

Again we expand the factors $P_e(\cdot,\cdot)$ using lemma 8.1. The factor $\pi/8$ does not change, and the squareroot-factor can again be lowerbounded by $1/\sqrt{2}$. As before observe that

$$\frac{(\frac{2^k}{2^k+b^*+1})^{2^k}(\frac{b^*+1}{2^k+b^*+1})^{b^*+1}}{(\frac{2^{k-1}}{2^{k-1}+\frac{b^*+1}{2}})^{2^{k-1}}(\frac{\frac{b^*+1}{2}}{2^{k-1}+\frac{b^*+1}{2}})^{\frac{b^*+1}{2}}} = (\frac{2^{k-1}}{2^{k-1}+\frac{b^*+1}{2}})^{2^{k-1}}(\frac{\frac{b^*+1}{2}}{2^{k-1}+\frac{b^*+1}{2}})^{\frac{b^*+1}{2}}. \tag{98}$$

The log-sum inequality yields

$$(\frac{2^{k-1}}{2^{k-1}+\frac{b^*+1}{2}})^{2^{k-1}}(\frac{a-2^{k-1}}{a-2^{k-1}+b-\frac{b^*-1}{2}})^{a-2^{k-1}} \geq (\frac{a}{a+b+1})^a. \tag{99}$$

For the remaining terms we find that

$$\begin{aligned}
(\frac{\frac{b^*+1}{2}}{2^{k-1}+\frac{b^*+1}{2}})^{\frac{b^*+1}{2}}(\frac{b-\frac{b^*-1}{2}}{a-2^{k-1}+b-\frac{b^*-1}{2}})^{b-\frac{b^*-1}{2}} \cdot \frac{\frac{b+\frac{1}{2}}{a+b+1}}{\frac{b^*+\frac{1}{2}}{2^k+b^*+1}} &\geq \\
(\frac{\frac{b^*+1}{2}}{2^{k-1}+\frac{b^*+1}{2}})^{\frac{b^*+1}{2}}(\frac{b-\frac{b^*-1}{2}}{a-2^{k-1}+b-\frac{b^*-1}{2}})^{b-\frac{b^*-1}{2}} \cdot \frac{\frac{b+1}{a+b+1}}{\frac{b^*+1}{2^k+b^*+1}} &\geq \\
(\frac{\frac{b^*+1}{2}}{2^{k-1}+\frac{b^*+1}{2}})^{\frac{b^*-1}{2}}(\frac{b-\frac{b^*-1}{2}}{a-2^{k-1}+b-\frac{b^*-1}{2}})^{b-\frac{b^*-1}{2}} \cdot \frac{b+1}{a+b+1} &\geq \\
(\frac{\frac{b^*-1}{2}}{2^{k-1}+\frac{b^*+1}{2}})^{\frac{b^*-1}{2}}(\frac{b-\frac{b^*-1}{2}}{a-2^k-1+b-\frac{b^*-1}{2}})^{b-\frac{b^*-1}{2}} \cdot \frac{b+1}{a+b+1} &\geq \\
(\frac{b}{a+b+1})^b\frac{b+1}{a+b+1} \geq (\frac{b}{b+1})^b(\frac{b+1}{a+b+1})^{b+1} &\geq \frac{1}{e}(\frac{b+1}{a+b+1})^{b+1}.
\end{aligned} \tag{100}$$

Combining all this we obtain for odd values of b^* that

$$\hat{P}_e(a,b) \geq \frac{\pi}{8e\sqrt{2}} \cdot P_e(a,b). \tag{101}$$

Together with (96) this implies that (101) holds for all b^*.

8.2. More rescaling. If, after the first rescaling operation, again one of the counts reaches 2^k, a second rescaling operation is necessary, etc. After r rescalings, this results in a coding distribution $\hat{P}_e^r(\cdot,\cdot)$ which can be expressed recursively as

$$\hat{P}_e^r(a,b) = \frac{P_e(a^r,b^r)\hat{P}_e^{r-1}(a-a_m^r,b-b_m^r)}{P_e(a^r-a_m^r,b^r-b_m^r)} \text{ for } a \geq a^r, b \geq b^r, \tag{102}$$

where $P_e^0(\cdot,\cdot) = P_e(\cdot,\cdot)$. Here a^r and b^r are the number of zeros and ones in the counters just *before* the r'th rescaling, and a_m^r and b_m^r the number of missing zero- and one-counts after the r'th rescaling. E.g. if we rescale for the first time to prevent the zero-count from overflowing, then $a^1 = 2^k$ and $b^1 = b^*$, and $a_m^1 = 2^{k-1}$ and $b_m^1 = \lfloor b^*/2 \rfloor$ (compare with (92) and observe that $b^* = \lceil b^*/2 \rceil + \lfloor b^*/2 \rfloor$). Note that $\hat{P}_e^r(a,b)$ actually depends not only on a and b but also on $a^1, a^2, \cdots, a^r$ and $b^1, b^2, \cdots, b^r$. By induction we can now prove that

$$\hat{P}_e^r(a,b) \geq (\frac{\pi}{8e\sqrt{2}})^r \cdot P_e(a,b). \tag{103}$$

8.3. Memoryless case performance. The question now arises how many rescalings there occur in a sequence of length T. Since the number of input symbols between subsequent rescaling operations can not be smaller than 2^{k-1}, the total number of rescalings is upperbounded by $T/2^{k-1}$.

If we use $\hat{P}_e^r(\cdot,\cdot)$ as coding distribution, i.e. $P_c(\cdots) = \hat{P}_e^r(\cdot,\cdot)$ with r increasing properly, then for the scaling redundancy we obtain

$$\log \frac{P_e(x_1^T)}{P_c(x_1^T)} \le r \cdot \log(\frac{8e\sqrt{2}}{\pi}) \le \frac{T}{2^{k-1}} \cdot \log(\frac{8e\sqrt{2}}{\pi}), \text{ for any } x_1^T. \tag{104}$$

The following equation shows that the total redundancy (forgetting coding redundancy for a moment) is the sum of the parameter redundancy and the scaling redundancy :

$$\log \frac{P_a(x_1^T)}{P_c(x_1^T)} = \log \frac{P_a(x_1^T)}{P_e(x_1^T)} + \log \frac{P_e(x_1^T)}{P_c(x_1^T)}. \tag{105}$$

8.4. Context tree weighting performance. Now consider a tree model (see [**48**]) with suffix set $\mathcal{S}$. The contents of the counters corresponding to the leaves in this model add up to T, i.e. the sequence length. Again the total number of rescalings, in all the counters corresponding to leaves of $\mathcal{S}$, is upperbounded by $T/2^{k-1}$. Therefore we conclude that also for *any* tree model the total scaling redundancy is upperbounded by

$$\log \frac{\Pi_{s\in\mathcal{S}} P_e(a_s, b_s)}{P_c^{\mathcal{S}}(x_1^T)} \le \frac{T}{2^{k-1}} \cdot \log(\frac{8e\sqrt{2}}{\pi}), \tag{106}$$

where $P_c^{\mathcal{S}}(\cdot)$ is the coding distribution composed out of rescaling distributions, one for each leaf of $\mathcal{S}$. Note that this amounts to 3.291 bits per 2^{k-1} symbols. This linear contribution can be made smaller by increasing the register-size k.

All this holds also for the context tree weighting algorithm, which was introduced at the 1993 ISIT by Willems, Shtarkov and Tjalkens [**56**]. If we use rescaling Dirichlet estimators there, we loose, no matter what the model is, not more than 3.291 bits per 2^{k-1} symbols.

8.5. Discussion. We should be aware of the fact that this bound is not tight for a number of reasons. First we should realize that for large models the actual number of rescalings can be much smaller than $T/2^{k-1}$ since there are many counters that can accommodate all the counts adding up to T. Secondly we should realize that the factor $8/\pi$ is present only since we use lemma 8.1, which holds for the whole range $a + b > 1$, whereas in our approach there are certain interesting restrictions on and relations between the parameters in the factors $P_e(\cdot,\cdot)$ in (92) that are not used yet. Using this extra information could lead to a vanishing contribution to the scaling redundancy of the factor which is now $8/\pi$. The squareroot factor, which is lowerbounded by $1/\sqrt{2}$, seems unavoidable. The same holds for the $1/e$-factor for odd values of b^*, however note that our bound is pessimistic in the sense that is also assumes the same loss for even values of b^*, which appear roughly half of the time. Combining all this, it is fair to assume that a rescaling operation actually costs $\log\sqrt{2e} = 1.221$ bits.

CHAPTER 6

Arithmetic Encoding and Decoding

1. Arithmetic Encoder and Decoder Implementation in CTW-1

We will start this chapter by giving a short description of the encoder and decoder implementation in the CTW-1 project. These coders are based on the Rubin[**30**] implementation of the Elias algorithm(see appendix 9 of chapter 3).

1.1. The Rubin Encoder. The idea is that an interval $I(x_1^{t-1})$ is represented by the integers $b(x_1^{t-1})$ and $p(x_1^{t-1})$, where $b(\cdot)$ denotes the last digits of the interval's lower boundary point and $p(\cdot)$ its size. Both $b(\cdot)$ and $p(\cdot)$ are stored in buffers containing f binary digits. More significant binary digits that determine the lower boundary point are assumed to be transmitted already. The encoder task can be subdivided in three different subtasks. We will shortly describe them here. After that we will give the program.

1.1.1. *Initialisation of the encoding process.* It will be clear that during the initialisation we have to achieve that $b(\phi) := 0$ and $p(\phi) := 2^f$. Here ϕ is the empty sequence.

1.1.2. *Scaling and subdividing.* Processing source symbol x_t starts with scaling $b(x_1^{t-1})$ and $p(x_1^{t-1})$. If it turns out that certain digits of $b(x_1^{t-1})$ will not change anymore they will be shifted out (and, they will be stored or transmitted). After scaling, the interval is such that $0 \leq b(x_1^{t-1}) < 2^{f-1} < b(x_1^{t-1}) + p(x_1^{t-1}) \leq 2^f$. Now it can be subdivided and according to the value of the source symbol x_t one of the subintervals will be taken for further processing.

1.1.3. *Termination of the encoding process.* After all symbols $x_1, x_2, \cdots, x_T$ are processed, we have to transmit or store $b(x_1^T)$ as last part of the codeword.

1.1.4. *The encoding program.* We now give the encoding program in 'pseudo-Pascal'. Note that we do not work with $p(x_1^t)$ but with $p(x_1^t) - 1$ instead.

```
b:=0; pmin1=2^f-1;                                    {initialisation}
FOR t:=1 TO T                                   {encoding source symbols}
DO BEGIN REPEAT IF b+pmin1<2^(f-1)               {scaling = transmitting}
                THEN BEGIN Push(0);b:=b*2;pmin1:=pmin1*2+1; END;
                IF b>=2^(f-1)
                THEN BEGIN Push(1);b:=(b-2^(f-1))*2;pmin1:=pmin1*2+1; END;
         UNTIL  ( b<2^(f-1) ) AND ( 2^(f-1)<=b+pmin1 );
         p0:=(pmin1*A_t+B_t) DIV (A_t+B_t);                 {subdividing}
         IF x_t=0 THEN pmin1:=p0-1
                  ELSE BEGIN b:=b+p0;pmin1:=pmin1-p0; END;
    END;
FOR i:=1 TO f                                              {terminating}
DO IF b<2^(f-1) THEN BEGIN Push(0);b:=b*2; END
```

```
ELSE BEGIN Push(1);b:=(b-2^(f-1))*2; END.
```

We assume that `A_t` and `B_t` have the same quotient as the conditional probabilities $P_c(X_t = 0|X^{t-1} = x_1^{t-1})$ and $P_c(X_t = 1|X^{t-1} = x_1^{t-1})$ respectively (or the block probabilities $P_c(X^{t-1} = x_1^{t-1}, X_t = 0)$ and $P_c(X^{t-1} = x_1^{t-1}, X_t = 1)$. The codeword digits are being transmitted by the function-calls `Push(0)` or `Push(1)`.

1.2. The Rubin Decoder. Without comments we give the decoder program.

1.2.1. *The decoding program.* This program is listed in 'pseudo-Pascal'. Note that `c` is the register containing the relevant code digits. These binary digits are received by the `Pull` function.

```
c:=0;b:=0;pmin1=2^f-1;                                   {initialisation}
FOR i:=1 TO f DO c:=2*c+Pull;
FOR t:=1 TO T                                      {decoding source symbols}
DO BEGIN REPEAT IF b+pmin1<2^(f-1)                    {scaling = receiving}
                THEN BEGIN c:=c*2+Pull;
                           b:=b*2;pmin1:=pmin1*2+1; END;
                IF b>=2^(f-1)
                THEN BEGIN c:=(c-2^(f-1))*2+Pull;
                           b:=(b-2^(f-1))*2;pmin1:=pmin1*2+1; END;
       UNTIL  ( b<2^(f-1) ) AND ( 2^(f-1)<=b+pmin1 );
       p0:=(pmin1*A_t+B_t) DIV (A_t+B_t);          {comparing & subdividing}
       IF c<b+p0 THEN BEGIN x_t:=0;pmin1:=p0-1; END
                 ELSE BEGIN x_t:=1;b:=b+p0;pmin1:=pmin1-p0; END;
   END.
```

1.3. Comments. It follows from both programs that the Rubin algorithm requires very little storage (only two or three f-bit registers are needed). On the other hand it is necessary to multiply and divide for processing a source symbol. This is a disadvantage. Furthermore a disadvantage is that the interval size $p(\cdot)$ can become very small (even 2 is possible). This makes it difficult to partition this interval into subintervals of the right relative size. The implementation that we will discuss next, does not have the the mentioned disadvantages. However we need more storage now.

2. Rissanen-Langdon Arithmetic Coding Approach

2.1. Introduction. In this section the special structure that is formed by a Context-Tree Weighting modeler combined with an arithmetic encoder or decoder, is investigated. The interaction between the CTW modeler and the Rubin coder as in CTW1 can be considered small compared to that of the CTW modeler and the coders that we will develop next.

Suppose that the CTW modeler produces block probabilities as described in [**57**] (see also chapter 3). These block probabilities are delivered to the arithmetic encoder and decoder. This has the advantage that the arithmetic encoder and decoder do not have to multiply to subdivide the source interval in a 0-interval and a 1-interval. This would

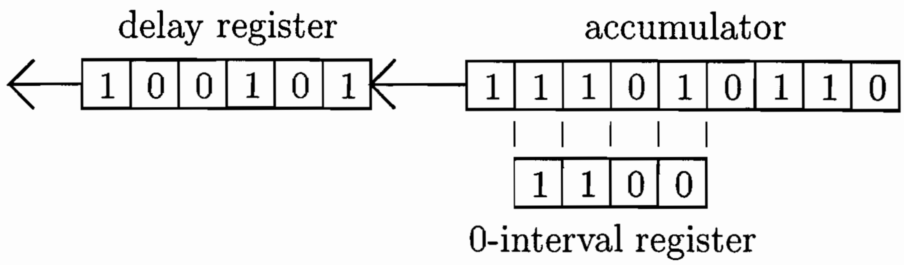


FIGURE 6.1. Arithmetic encoding and decoding structure.

have been necessary if the modeler would produce conditional probabilities as usual. The disadvantage of this method is however that we have to be very careful when we define the finite accuracy operations done in the context tree (see Willems[**54**]).

We will use the notation of [**57**] and chapter 3 to study the arithmetic encoder and decoder.

2.2. Floating point numbers. The context tree weighting method supplies block probabilities P_c to the arithmetic coder or decoder. These coding probabilities are assumed to satisfy[1]

$$\begin{aligned} P_c(\phi) &= 1, \\ P_c(x_1^{t-1}) &\geq P_c(x_1^{t-1}, X_t = 0) + P_c(x_1^{t-1}, X_t = 1), \\ &\quad \text{for all } x_1^{t-1}, \text{ and } t = 1, \cdots, T, \text{ and} \\ P_c(x_1^T) &> 0, \text{ for all } x_1^T. \end{aligned} \tag{107}$$

In addition to this, the coding probabilities $P_c(x_1^t), t = 0, T$ are assumed to be represented by f-bit *floating point numbers.* An f-bit floating point number g can be written asy

$$g = m(g) \cdot 2^{e(g)}, \text{ with } 2^{f-1} \leq m(g) < 2^f, \tag{108}$$

where the mantissa $m(g)$ and the exponent $e(g)$ are integer-valued. Note that storing a floating point number requires f binary positions for the mantissa[2] and an additional number of positions for the exponent.

2.3. Encoder structure. Consider an arithmetic encoding and decoding structure as in Figure 6.1. With this structure we intend to compute $B(x_1^T)$, i.e. the lower boundary point of the interval $I(x_1^T)$ that corresponds to the sequence x_1^T. Note (see e.g. (36)) that this can be done recursively:

$$\begin{aligned} B(x_1^t) &= \sum_{\tilde{x}_1^t < x_1^t} P_c(x_1^t) \\ &= \sum_{\tilde{x}_1^{t-1} < x_1^{t-1}} P_c(\tilde{x}_1^{t-1}) + \sum_{\tilde{x}_t < x_t} P_c(x_1^{t-1}, \tilde{x}_t) \\ &= B(x_1^{t-1}) + \sum_{\tilde{x}_t < x_t} P_c(x_1^{t-1}, \tilde{x}_t), \end{aligned} \tag{109}$$

[1]Here, in contrast with [**57**], we do not require these 'probabilities' to sum up to 1, and since all source sequences can occur we want all probabilities to be positive.

[2]Also $f - 1$ positions would be enough.

where we assume that $B(\phi) = 0$. Therefore we are adding, only if $X_t = 1$, the term $P_c(x_1^{t-1}, X_t = 0)$, which is placed in the 0-interval register, to $B(x_1^{t-1})$, which is stored in the accumulator.

When we are adding terms to an accumulator, carries resulting from these operations can work their way up to the most-significant positions. Therefore in principle we have to keep the complete $B(x_1^{t-1})$ in this accumulator. To avoid this, we compute instead of $B(x_1^T)$ its 'approximation' $\tilde{B}(x_1^T)$. We do this by using a *delay register* as suggested by Rissanen and Langdon (see [**29**]). The structure in Figure 6.1 contains a number of binary digits of $B(x_1^{t-1})$ before the source symbol x_t is about to be processed.

If $P_c(x_1^{t-1}) = m(x_1^{t-1}) \cdot 2^{e(x_1^{t-1})}$ the most significant position of the accumulator contains digit $1 - f - e(x_1^{t-1}) + \Omega(x_1^{t-1})$ of $\tilde{B}(x_1^{t-1})$. The delay register contains more-significant digits of $B(x_1^{t-1})$, at least one of them is assumed to be zero. The *shift number* $\Omega(x_1^{t-1})$ depends on the previously processed source symbols x_1^{t-1}. This shift number, which is initially 0, will be incremented occasionally as we will soon see.

In the 0-interval register (which is f positions wide), we now store the mantissa of $P_c(x_1^{t-1}, X_t = 0) = m(x_1^{t-1}, 0) \cdot 2^{e(x_1^{t-1},0)}$. Depending on the actual value x_t the 0-interval register is added to the accumulator, i.e. we compute

$$\tilde{B}(x_1^t) = \tilde{B}(x_1^{t-1}) + 2^{-\Omega(x_1^{t-1})} \cdot \sum_{\tilde{x}_t < x_t} P_c(x_1^{t-1}, \tilde{x}_t). \tag{110}$$

If $P_c(x_1^{t-1}, X_t = 0)$ is added, the 0-interval register is aligned with the accumulator. In the figure it is assumed that the $e(x_1^{t-1}, 0) = e(x_1^{t-1}) - 1$. Note that $P_c(x_1^{t-1}, 0) = m(x_1^{t-1}, 0) \cdot 2^{e(x_1^{t-1},0)}$ determines $e(x_1^{t-1}, 0)$.

Because of the restrictions on the coding distribution and the fact that the shift number will never decrease, the sum of all that can be added to the accumulator from now on, can not be more than $2^{-\Omega(x_1^{t-1})} P_c(x_1^{t-1}) < 2^{-\Omega(x_1^{t-1})} 2^{f+e(x_1^{t-1})} = 2^{f+e(x_1^{t-1})-\Omega(x_1^{t-1})}$. The content of the accumulator corresponds to a value which is also less than $2^{f+e(x_1^{t-1})-\Omega(x_1^{t-1})}$. Therefore not more than one carry (i.e. $2^{f+e(x_1^{t-1})-\Omega(x_1^{t-1})}$) can flow out of the accumulator into the delay register. Since it was assumed that the delay register contains at least one zero, this carry can always be accommodated by the delay register such that the delay register does not have to produce a carry itself.

After having added $P_c(x_1^{t-1}, X_t = 0)$ to the accumulator (or not), the contents of the accumulator and delay register are shifted to the left, if neccessary, until the most-significant position of the accumulator contains digit $1 - f - e(x_1^{t-1}, x_t) + \Omega(x_1^{t-1})$ of $\tilde{B}(x_1^{t-1}, x_t)$. Now the delay register is checked. If this register does not contain a zero, we increment the shift number Ω and shift the contents of the accumulator and delay register over one position to the left. We keep doing this until the delay register contains at least one zero. Since we are shifting in zeros at the least-significant end of the accumulator, this will always happen finally. The value that Ω has reached now, is denoted by $\Omega(x_1^{t-1}, x_t)$. Now the next source symbol can be processed.

This is how the encoder computes $\tilde{B}(x_1^T)$. In practise this $\tilde{B}(x_1^T)$ will serve as the codeword. This means that digits (corresponding to exponents $-1, -2, \cdots$ etc.) that leave the delay register, can already be transmitted to the decoder during encoding. After having processed the last symbol x_T no left shifting is necessary[3] We stop with digit $1-f-e(x_1^{T-1})+\Omega(x_1^{T-1})$ in the most-significant position of the accumulator. Now these contents of the delay register and accumulator form the last part of the codeword.

2.4. Decoder structure. How does the decoder operate? Just like the encoder, before decoding symbol x_t, the accumulator of the decoder contains digit $1-f-e(x_1^{t-1})+\Omega(x_1^{t-1})$ of $\tilde{B}(x_1^{t-1})$ on the most-significant position (and the delay register contains at least one zero). Again in the 0-interval register, we store the mantissa of $P_c(x_1^{t-1}, X_t=0) = m(x_1^{t-1},0)\cdot 2^{e(x_1^{t-1},0)}$. Now the threshold $\tilde{D}(x_1^{t-1})$ is computed by adding this mantissa (after appropriate aligning) to the accumulator, i.e.

$$\tilde{D}(x_1^{t-1}) = \tilde{B}(x_1^{t-1}) + 2^{-\Omega(x_1^{t-1})}P_c(x_1^{t-1}, X_t=0). \tag{111}$$

Note that a carry can result from this addition. No left-shifting is done at this moment however! Unique decodability of source symbol x_t is now guaranteed if for the threshold $\tilde{D}(x_1^{t-1}) = \tilde{B}(x_1^{t-1}, X_t = 1)$ we can show that $\tilde{B}(x_1^{t-1}, X_t=0, x_{t+1}^T) < \tilde{D}(x_1^{t-1}) \leq \tilde{B}(x_1^{t-1}, X_t=1, x_{t+1}^T)$ for all $x_{t+1}^T \in \{0,1\}^{T-t}$. The second inequality is obviously true. The first one holds since

$$\begin{aligned}
&\tilde{B}(x_1^{t-1}, X_t=0, x_{t+1}^T) \\
&\quad\leq \tilde{B}(x_1^{t-1}, X_tX_{t+1}\cdots X_T = 01\cdots 1) \\
&\quad= \tilde{B}(x_1^{t-1}) + \sum_{\tau=t+1,T} 2^{-\Omega(x_1^{t-1}, X_tX_{t+1}\cdots X_{\tau-1}=01\cdots 1)}P_c(x_1^{t-1}, X_tX_{t+1}\cdots X_{\tau-1}X_\tau = 01\cdots 10) \\
&\quad\leq \tilde{B}(x_1^{t-1}) + 2^{-\Omega(x_1^{t-1})}\cdot \sum_{\tau=t+1,T} P_c(x_1^{t-1}, X_tX_{t+1}\cdots X_{\tau-1}X_\tau = 01\cdots 10) \\
&\quad< \tilde{B}(x_1^{t-1}) + 2^{-\Omega(x_1^{t-1})}P_c(x_1^{t-1}, X_t=0) \\
&\quad= \tilde{D}(x_1^{t-1}).
\end{aligned} \tag{112}$$

First note that the last inequality in (112) is strict since all coding probabilities, and hence also $P_c(x_1^{t-1}, X_tX_{t+1}\cdots X_T = 01\cdots 1)$, are assumed to be positive[4]. Furthermore note that the threshold $\tilde{D}(x_1^{t-1})$ is finite, i.e. does not have less-significant non-zero digits outside the accumulator. Therefore and since the codeword falls within the correct range, i.e.[5] $\tilde{B}(x_1^{t-1}) \leq \tilde{B}(x_1^{t-1}, x_t^T) < \tilde{B}(x_1^{t-1}) + 2^{-\Omega(x_1^{t-1})}P_c(x_1^{t-1})$ for any $x_t^T \in \{0,1\}^{T-t+1}$ it is possible to compare the codeword to this threshold and to check whether it is greater than or equal to *or alternatively* smaller than the threshold, *within* the delay register and accumulator only.

2.5. Concluding remarks. The carry blocking procedure that we have described here occurs in Tjalkens' Ph.D. thesis [42]). It will be clear that the effect of these shifting operations is that the codeword-length increases. It is not so easy to find an upperbound on the number of shifts that can occur, however assuming that $\tilde{B}(x_1^t)$ before checking the delay register consists of uniformly distributed binary digits, yields a probability of 2^{-d}

[3]An other way of looking at this is that procesing a symbol starts with scaling and checking the delay register. After that the interval is subdivided.

[4]If $P_c(x_1^{t-1}, X_tX_{t+1}\cdots X_T = 01\cdots 1)$ is not positive, replace $01\cdots 1$ in the argument by the largest sequence x_{t+1}^T in lexicographical sense, such that $P_c(x_1^{t-1}, X_t=0, x_{t+1}^T) > 0$.

[5]Again we assume here that $P_c(x_1^{t-1}, X_tX_{t+1}\cdots X_T = 11\cdots 1) > 0$.

that a first shift is necessary if the delay register has size C. Each additional shift is now necessary with probability 1/2, yielding 2 shifts in total on the average, and an expected number of 2^{1-d} shifts per processed source symbol. Taking d equal to 16 or more then gives a negligable redundancy increase. These considerations are confirmed by experiments (see again [**42**]).

To the delay register we must add a carry occasionally. The accumulator is more complex than the delay register since it must be possible to add a floating point number to it. The size of the accumulator should in principle be quite large if the coding probability $P_c(x_1^{t-1}, X_t = 0)$ of the 0-extension is considerably smaller than $P_c(x_1^{t-1})$. However it always is possible (see [**16**]) to transform the symbols zero and one into most-probable symbol (α_+) and least-probable symbol (α_-), and define the ordering $\alpha_+ < \alpha_-$ for arithmetic coding. This has the effect that the interval register now contains the α_+-probability, which is not less than half the previous interval size. Consequently if we align the interval register to the accumulator the most-significant position of the interval register must match with the most-significant position of the accumulator or with the position to the right of this most-significant position. Therefore the size of the accumulator need not be more than $f+1$ positions. An additional advantage of this approach is that the probability of having to add (and this adding increases the processing time), is minimized.

Now that we know how to encode and decode with finite accuracy if the CTW-modeler produces block probabilities $P_c(x_1^{t-1}, x_t)$ that satisfy (107), we will change this setup a little bit in the next subsection. We will assume there that the modeler produces instead of block probabilities, conditional probabilities $P_c(x_t|x_1^{t-1})$.

3. From Block to Conditional Probabilities

Suppose we have a coding situation as described in the previous section where the modeler supplies the encoder and decoder with block probabilities. More exactly, if the encoder and decoder have already processed x_1^{t-1} they know $P_c(x_1^{t-1})$ and for processing the next symbol x_t the block probabilities $P_c(x_1^{t-1}, X_t = 0)$ and $P_c(x_1^{t-1}, X_t = 1)$ are made available to the encoder and the decoder.

If the encoder and decoder would get some information from the modeler to compute $P_c(x_1^{t-1}, X_t = 0)$ and $P_c(x_1^{t-1}, X_t = 1)$ themselves out of $P_c(x_1^{t-1})$, the situation would be as before. What information should the modeler give to the coders? The answer to this question is simple. The ratios $P_c(x_1^{t-1}, X_t = 0)/P_c(x_1^{t-1})$ and $P_c(x_1^{t-1}, X_t = 1)/P_c(x_1^{t-1})$ suffice to do this. These ratios are conditional probabilities if $P_c(x_1^{t-1}, X_t = 0) + P_c(x_1^{t-1}, X_t = 1) = P_c(x_1^{t-1})$.

Now suppose that after having processed x_1^{t-1} the current interval has size $S(x_1^{t-1}) = 2^{-\Omega(x_1^{t-1})} \cdot P_c(x_1^{t-1})$. With the "conditional probability" $P_c(X_t = 0|x_1^{t-1})$ the encoder and decoder can compute the size of the 0-subinterval $S(x_1^{t-1}, X_t = 0)$. Note that some accuracy can be lost here and $S(x_1^{t-1}, X_t = 0)$ can be different from what the modeler had in mind. Instead of using the "conditional probability" $P_c(X_t = 1|x_1^{t-1})$ to calculate $S(x_1^{t-1}, X_t = 0)$, the encoder and decoder can set $S(x_1^{t-1}, X_t = 1)$ equal to the highest possible value $S(x_1^{t-1}) - S(x_1^{t-1}, X_t = 0)$. This assignment gives the lowest possible redundancy (given $S(x_1^{t-1}, X_t = 0)$ of course).

Now suppose that 0 is the, or a, symbol with highest probability (if not, we have to relabel) then $S(x_1^{t-1}, X_t = 0) \geq S(x_1^{t-1}, X_t = 1)$. Remember that both $S(x_1^{t-1})$ and

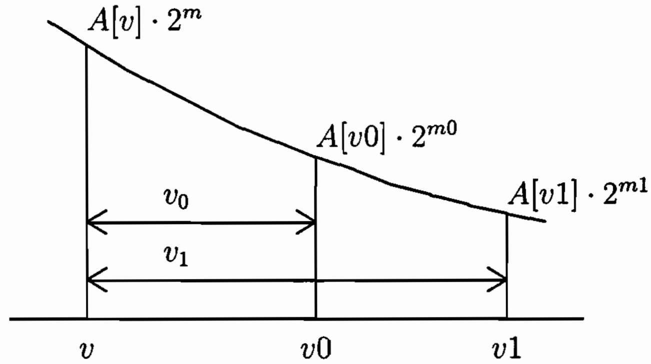


FIGURE 6.2. Splitting up $A[v] \cdot 2^m$ in $A[v0] \cdot 2^{m0}$ and $A[v1] \cdot 2^{v1}$.

$S(x_1^{t-1}, X_t = 0)$ are f-bit floating point numbers, i.e. $S(x_1^{t-1}) = s \cdot 2^m$ and $S(x_1^{t-1}, X_t = 0) = s0 \cdot 2^{m0}$, where both s and $s0$ are integers within the range $\{2^{f-1}, 2^{f-1}+1, \cdots, 2^f - 1\}$. Now

$$\begin{aligned} S(x_1^{t-1}, X_t = 1) &= s \cdot 2^m - s0 \cdot 2^{m0} \\ &= (s \cdot 2^{m-m0} - s0) \cdot 2^{m0} \\ &\leq s0 \cdot 2^{m0}, \end{aligned} \tag{113}$$

where $s \cdot 2^{m-m0} - s0$ is integer also since $m \geq m0$. If $S(x_1^{t-1}, X_t = 1) = s1 \cdot 2^{m1}$ with $s \cdot 2^{m-m0} - s0$ and $m1 = m0$, and we multiply $s1$ by 2 and decrement $m1$ as long as $s1 < 2^{f-1}$, the interval $S(x_1^{t-1}, X_t = 1)$ can finally be represented as an f-bit floating number too.

We can assume that the modeler in principle is able to deliver block probabilities that satisfy (107) as described in Willems [**54**]. Then this modeler has a guaranteed performance. However there can be some loss of accuracy by forming the conditional probability $P_c(X_t = 0|x_1^{t-1})$. This loss could be bounded if the truncation operation needed to form the conditional probability is well specified and not to coarse. In practise this is not very useful however. Moreover it is not necessary to design the modeler such that it could deliver block probabilities that satisfy (107). We are not interested in a guaranteed performance if it will increase the complexity a lot.

In the CTW-2 project we only want the modeler to generate conditional probabilities $P_c(X_t = 0|x_1^{t-1})$ that give good results in our simulations. Only in this way we are able to achieve a reasonable complexity of our algorithm.

4. Exponential Tables and Stepsizes

Since multiplying is a more complex operation than adding, it is advantageous to use exp- and log-tables to determine the size $S(x_1^{t-1}, X_t = 0)$ from $S(x_1^{t-1})$ and the conditional probability $P_c(X_t = 0|x_1^{t-1})$. This idea can be found already in Rissanen[**24**] in a LIFO algorithm. It and was investigated in more detail (and for a FIFO method) in Tjalkens and Willems[**44**] and in Tjalkens' Ph.D. thesis[**42**].

4.1. The exp-table $A[\cdot]$. What we mean is the following. Suppose we have an exp-table

$$A[i] := \lfloor 2^f \cdot 2^{-i/2^f} + 1/2 \rfloor \text{ for } i = 1, 2, \cdots, 2^f, \tag{114}$$

where $\lfloor a \rfloor$ is the largest integer not larger than a. First of all we want the numbers $A[i]$ for $i = 1, 2^f$ to be f-bit floating point numbers since they will be used to represent interval sizes $S(x_1^t)$ for $t = 0, T$. E.g. (see Figure 6.2)

$$S(x_1^{t-1}) = A[v] \cdot 2^m \text{ for some } v \in \{1, 2, \cdots, 2^f\} \text{ and integer } m, \tag{115}$$

so $S(x_1^{t-1})$ is represented by the pair (v, m). The table values are not smaller than 2^{f-1}. They are all f-bit floating numbers if the largest table value $A[1] < 2^f$. Note that

$$\begin{aligned} 2^f \cdot 2^{-1/2^f} &< 2^f \cdot (1 - \frac{1}{2^{f+1}}) \\ &= 2^f - 1/2. \end{aligned} \tag{116}$$

The (strict) inequality follows from the fact that $2^{-t} < 1 - t/2$ for $0 < t < 1$. Therefore we get the strict inequality

$$A[1] = \lfloor 2^f \cdot 2^{-1/2^f} + 1/2 \rfloor < 2^f. \tag{117}$$

Now the interval size $S(x_1^{t-1}, X_t = 0)$ is determined as follows. Suppose that instead of the conditional probability $P_c(X_t = 0 | x_1^{t-1})$ the coders receive a *stepsize* v_0 which is defined as

$$v_0 = \lfloor 2^f \cdot \log \frac{1}{P_c(X_t = 0 | x_1^{t-1})} + 1/2 \rfloor. \tag{118}$$

Since 0 is the(a) most probable symbol, which is always the case after relabeling, we may assume that $v_0 \le 2^f$. This stepsize is now used to find $S(x_1^{t-1}, X_t = 0)$ by adding this v_0 to v. More precisely

$$S(x_1^{t-1}, X_t = 0) = \begin{cases} A[v + v_0] \cdot 2^m & \text{if } v + v_0 \le 2^f, \\ A[v + v_0 - 2^f] \cdot 2^{m-1} & \text{if } v + v_0 > 2^f. \end{cases} \tag{119}$$

Note that always $v + v_0 \le 2 \cdot 2^f$ and thus $v + v_0 - 2^f \le 2^f$. So also $S(x_1^{t-1}, X_t = 0)$ can be represented by a pair $(v0, m0)$ as in (119). See again Figure 6.2.

Next we want the table to be such that the numbers $2^{f-1}, 2^{f-1} + 1, \cdots, 2^f - 1$ all exist as table values. The reason for this is that the subtraction that is needed to compute $S(x_1^{t-1}, X_t = 1)$ always will result in an f-bit floating point number. This follows from (113). Note that

$$S(x_1^{t-1}, X_t = 1) = S(x_1^{t-1}) - S(x_1^{t-1}, X_t = 0) = A[v] \cdot 2^m - A[v0] \cdot 2^{m0}. \tag{120}$$

We would like to express this interval size $S(x_1^{t-1}, X_t = 1)$ by a pair $(v1, m1)$ such that

$$S(x_1^{t-1}, X_t = 1) = A[v1] \cdot 2^{m1}, \tag{121}$$

for some integer $v1$ and some $m1$. See Figure 6.2. This search can be carried out without problems if the $A[i]$ for $i = 1, 2^f$ together cover all f-bit values $2^{f-1}, 2^{f-1} + 1, \cdots, 2^f - 1$.

All f-bit values are covered if two "adjacent" table values do not differ more than 1. This is the case if $2^f \cdot 2^{-(i-1)/2^f} - 2^f \cdot 2^{-i/2^f} \leq 1$ for all $i = 1, 2^f$. Indeed for such i

$$\begin{aligned} 2^f \cdot 2^{-(i-1)/2^f} - 2^f \cdot 2^{-i/2^f} &= 2^f \left(2^{-(i-1)/2^f} - 2^{-i/2^f}\right) \\ &= \frac{2^{-(i-1)/2^f} - 2^{-i/2^f}}{1/2^f} \\ &\leq -\frac{d2^{-t}}{dt}\bigg|_{0\leq t\leq 1} = \ln(2)\cdot 2^{-t}\bigg|_{0\leq t\leq 1} = \ln(2) < 1. \quad (122) \end{aligned}$$

4.2. The log-table $B[\cdot]$. To make the search that is described at the end of the previous subsection easy, we construct a log-table $B[\cdot]$ which is the inverse of the table $A[\cdot]$. From (113) we know that initially

$$S(x_1^{t-1}, X_t = 1) = (A[v]\cdot 2^{m-m0} - A[v0])\cdot 2^{m0} \leq A[v0]\cdot 2^{m0}. \quad (123)$$

We now will use $A[v]\cdot 2^{m-m0} - A[v0]$ as entry in the $B[\cdot]$ table. Since $0 < A[v0] \leq 2^f - 1$, the table $B[j]$ must have entries for $j = 1, 2^f - 1$. The entries $B[j]$ for $j = 2^{f-1}, 2^f - 1$ are constructed from the exp-table $A[\cdot]$. Suppose that $\mathcal{I}$ is the set of i's such that $A[i] = j$. Note that $\mathcal{I}$ cannot be empty since all values $2^{f-1}, 2^{f-1}+1, \cdots, 2^f - 1$ exist as table values. Then we choose for the inverse $B[j]$ the i that minimizes the difference

$$|-2^f \log \frac{j}{2^f} - i|. \quad (124)$$

Note that in this way we obtain that always $1 \leq B[j] \leq 2^f$.

For the entries $j = 1, 2^{f-1} - 1$ we follow a different procedure. First we determine the exponent k such that $2^{f-1} \leq 2^k \cdot j \leq 2^f - 1$. Then we set $B[j] = B[2^k \cdot j] + k \cdot 2^f$. Since $1 \leq B[2^k \cdot j] \leq 2^f$ the exponent k can easily be found from $B[j]$. The number of bits that are needed to accomodate the table values $B[\cdot]$ is $f + \lceil \log f \rceil$ since k is at most $f - 1$ (when $j = 1$). Here $\lceil a \rceil$ is the smallest integer not smaller than a.

Note that with this table $B[\cdot]$ we can now find $v1$ and $m1$ such that $S(x_1^{t-1}, X_t = 1) = v1 \cdot 2^{m1}$ quite easily. We first set $v1 = B[A[v]\cdot 2^{m-m0} - A[v0]]$ and $m1 = m0$. Then we subtract 2^f from $v1$ and decrement $m1$ as long as $v1 > 2^f$. Finally $1 \leq v1 = B[2^k \cdot j] \leq 2^f$.

4.3. The minimum stepsize. Now we are almost finished. There is one thing left to discuss however. From (118) we see that in principle $0 \leq v_0 \leq 2^f$. However if v_0 is small then $A[v0]\cdot 2^{m0}$ can be equal to $A[v]\cdot 2^v$ and consequently the intervalsize $S(x_1^{t-1}, X_t = 1) = 0$. If the symbol $X_t = 1$ now occurs we are in trouble. Therefore we do not allow the stepsize v_0 to be smaller than 3. Then it is guaranteed that always $A[v]\cdot 2^m > A[v0]\cdot 2^{m0}$.

To see this first consider the case where v is such that the sum $v + v_0 > 2^f$. In that case $m0 = m - 1$ and since $A[v0] \leq 2^f - 1$ we obtain that $2\cdot A[v] \geq 2\cdot 2^{f-1} = 2^f > 2^f - 1 \geq A[v0]$ or $A[v]\cdot 2^m > A[v0]\cdot 2^{m0}$.

In the case where $v \geq 1$ is such that $v + v_0 \leq 2^f$ we get that $m0 = m$ and

$$\begin{aligned} 2^f \cdot 2^{-v/2^f} - 2^f \cdot 2^{-(v+v_0)/2^f} &= 2^f \left(2^{-v/2^f} - 2^{-(v+v_0)/2^f}\right) \\ &= v_0 \frac{2^{-v/2^f} - 2^{-(v+v_0)/2^f}}{v_0/2^f} \\ &\geq v_0 \min_{0\leq t\leq 1} -\frac{d2^{-t}}{dt} = v_0 \min_{0\leq t\leq 1} \ln(2)\cdot 2^{-t} = \frac{v_0 \ln(2)}{2} \\ &\geq \frac{3\ln(2)}{2} > 1. \end{aligned} \tag{125}$$

By the definition (114) of $A[\cdot]$ this implies that $A[v] > A[v + v_0]$. Since $v + v_0 = v0$ and $m0 = m$ we get that $A[v] \cdot 2^m > A[v0] \cdot 2^{m0}$.

5. Program Descriptions

5.1. The encoder program.

5.1.1. *Initialization.* The delay register `dlreg` is supposed to be d binary digits wide. The accumulator `accum` is assumed to be $f + 1$ binary digits wide. Both are set equal to zero. The interval size $S(\phi)$ is set equal to $A[1] = 2^f - 1$ by setting `v:=1`. Note that we do not set `v` equal to 0. The exponent m does not matter.

A pseudo-PASCAL program would look like:

```
dlreg:=0;
accum:=0;
v:=1;                                              {Initialization}
```

During the initialization also the tables(arrays) `A[ ]` and `B[ ]` are computed.

5.1.2. *Processing a symbol.* `Push` is the function that writes a binary digit to the codefile. Processing a symbol consists of three parts.

First it is checked whether the current interval, determined by `v`, can be scaled. This is the case if `v>2^f`. Each scaling operation results in a left-shift of both the delay register and the accumulator. The most significant bit of the accumulator is shifted into the delay register. The most significant bit of the delay register is `push`ed into the codefile. Then `v` is decreased by `2^f` and again it is checked whether the interval can be scaled. After scaling `1<=v<=2^f` and the current interval size has its most significant bit aligned with the most significant bit of the accumulator.

After having finished scaling, the delay register is checked. As long as it contains only 1's, both the delay register and the accumulator are shifted to the left, just as before, when we were scaling. The difference is now however that `v` does not change. The effect of this is that the current interval size is decreased by a factor of 2, in other words the shift number is incremented.

Now the encoder is ready to update the accumulator (and the delay register). It is assumed that `v_0` is not smaller than 3. Then the 0-interval size is determined by computing `v0=v+v_0`. Since `v0<=2^f` the size of this 0-interval can never be less than half the entire interval. Therefore the most significant bit of the 0-interval is aligned with the most significant bit of the accumulator (if v0<=2^f) or with the digit to the right of the most significant bit (if v0>2^f).

```
WHILE v>2^f
DO BEGIN
     IF dlreg>=2^(d-1) THEN BEGIN Push(1);dlreg:=2*(dlreg-2^(d-1)); END
                       ELSE BEGIN Push(0);dlreg:=2*dlreg; END;
     IF accum>=2^f THEN BEGIN dlreg=delreg+1;accum:=2*(accum-2^f); END
                   ELSE accum:=2*accum;
     v:=v-2^f;
   END;                                                 {Scaling and pushing}

WHILE dlreg=2^d-1
DO BEGIN
     IF dlreg>=2^(d-1) THEN BEGIN Push(1);dlreg:=2*(dlreg-2^(d-1)); END
                       ELSE BEGIN Push(0);dlreg:=2*dlreg; END;
     IF accum>=2^f THEN BEGIN dlreg=dlreg+1;accum:=2*(accum-2^f); END
                   ELSE accum:=2*accum;
   END;                                   {Creating zeroes in delay register}

v0=v+v_0;
IF X_t=1
THEN BEGIN IF v0<=2^f
           THEN BEGIN
                  accum=accum+2*A[v0];
                  IF accum>=2^(f+1)
                  THEN BEGIN dlreg:=dlreg+1;accum:=accum-2^(f+1); END
                  v:=B[A[v]-A[v0]];
                END
           ELSE BEGIN
                  accum=accum+A[v0-2^f];
                  IF accum>=2^(f+1)
                  THEN BEGIN dlreg:=dlreg+1;accum:=accum-2^(f+1); END
                  v:=B[2*A[v]-A[v0-2^f]]+2^f;
                END;
     END
ELSE v:=v0;     {Adding A[v0] to the accumulator (or not) and computing v}
```

5.1.3. *Termination.* The encoder terminates the coding process by Pushing the delay register and the accumulator.

```
FOR i:=1 TO d DO
IF dlreg<2^(d-1) THEN BEGIN Push(0);dlreg:=dlreg*2; END
                 ELSE BEGIN Push(1);dlreg:=(dlreg-2^(d-1))*2; END;
FOR i:=1 TO f+1 DO
```

```
IF accum<2^f THEN BEGIN Push(0);accum:=accum*2; END
             ELSE BEGIN Push(1);accum:=(accum-2^f)*2; END;   {Termination}
```

5.2. The decoder program.

5.2.1. *Initialization.* Both the delay register `dlreg` and the accumulator `accum` are set equal to zero. The interval size $S(\phi)$ is set equal to $A[1] = 2^f - 1$ by setting `v:=1`.

In the decoder we use an additional "code-delay register `cdlreg` and an additional "code-accumulator" `caccum`. The code-delay register is set equal to zero. Then, by calling the function `Pull` $f + 1$ times we fill the code-accumulator with code-bits. The code-bits flow through the code-accumulator and the code-delay register.

A pseudo-PASCAL program would look like:

```
dlreg:=0;
accum:=0;
v:=1;
cdlreg:=0;
caccum:=0;
FOR i:=1 TO f+1 DO caccum:=caccum*2+Pull;                  {Initialization}
```

During the initialization again the tables(arrays) `A[ ]` and `B[ ]` are computed.

5.2.2. *Decoding and processing a symbol.* Processing a symbol consists of four parts in the decoder, scaling, checking the delay register, decoding and updating the delay register and accumulator.

Just as in the encoder it is first checked whether the current interval, determined by `v`, can be scaled. This is the case if `v>2^f`. Each scaling operation results in a left-shift of both the delay register and the accumulator *and* the code-delay register and the code-accumulator. The most significant bits of the delay registers are shifted out. The most significant bits of the accumulators are shifted into the corresponding delay registers. With a `Pull` a code bit is recovered. This bit is shifted into the code-accumulator. After scaling `1<=v<=2^f` and the current interval size has its most significant bit aligned with the most significant bit of the accumulator.

After having finished scaling, the delay register is checked. As long as it contains only 1's, both the delay register and the accumulator *and* the code-delay register and the code-accumulator are shifted to the left, just as before, when we were scaling. Note that for each left-shift we again `Pull` a code bit. The difference is now however that `v` does not change. The effect of this is that the current interval size is decreased by a factor of 2, in other words the shift number is incremented.

Now the decoder is ready to decode and process the symbol `x_t`. First the threshold is computed. This threshold has both a delay register part and an accumulator part. We call them threshold-delay register(`tdlreg`) and threshold-accumulator(`taccum`). Now the code-delay register, accumulator pair is compared to the threshold-delay register, accumulator pair. The result determines the output `x_t`. With this result the decoder updates the accumulator and the delay register just like the encoder.

```
WHILE v>2^f
DO BEGIN
     IF dlreg>=2^(d-1) THEN dlreg:=2*(dlreg-2^(d-1));
                       ELSE dlreg:=2*dlreg;
     IF accum>=2^f THEN BEGIN dlreg=dlreg+1;accum:=2*(accum-2^f); END
                   ELSE accum:=2*accum;
     v:=v-2^f;
     IF cdlreg>=2^(d-1) THEN cdlreg:=2*(cdlreg-2^(d-1));
                        ELSE cdlreg:=2*cdlreg;
     IF caccum>=2^f THEN BEGIN cdlreg=cdlreg+1;
                               caccum:=2*(caccum-2^f)+Pull; END
                    ELSE caccum:=2*caccum+Pull;
   END;                                             {Scaling and pulling}

WHILE dlreg=2^d-1
DO BEGIN
     IF dlreg>=2^(d-1) THEN dlreg:=2*(dlreg-2^(d-1));
                       ELSE dlreg:=2*dlreg;
     IF accum>=2^f THEN BEGIN dlreg=dlreg+1;accum:=2*(accum-2^f); END
                   ELSE accum:=2*accum;
     IF cdlreg>=2^(d-1) THEN cdlreg:=2*(cdlreg-2^(d-1));
                        ELSE cdlreg:=2*cdlreg;
     IF caccum>=2^f THEN BEGIN cdlreg=cdlreg+1;
                               caccum:=2*(caccum-2^f)+Pull; END
                    ELSE caccum:=2*caccum+Pull;
   END;                                 {Creating zeroes in delay register}

v0=v+v_0;
IF v0<=2^f
THEN BEGIN
        taccum:=accum+2*A[v0];
        tdlreg:=dlreg;
        IF taccum>=2^(f+1)
        THEN BEGIN tdlreg:=tdlreg+1;taccum:=taccum-2^(f+1); END;
        IF ((cdlreg=tdlreg) AND (caccum<taccum))
        THEN x_t:=0 ELSE x_t:=1;                           {Decoding x_t}
        IF (x_t=1) THEN BEGIN accum=taccum;dlreg=tdlreg;
                              v:=B[A[v]-A[v0]]; END
                   ELSE v:=v0;
     END        {Adding A[v0] to the accumulator (or not) and computing v}
ELSE BEGIN
        taccum:=accum+A[v0-2^f];
        tdlreg:=dlreg;
        IF taccum>=2^(f+1)
        THEN BEGIN tdlreg:=tdlreg+1;taccum:=taccum-2^(f+1); END;
        IF ((cdlreg=tdlreg) AND (caccum<taccum))
```

```
        THEN x_t:=0 ELSE x_t:=1;                            {Decoding x_t}
        IF (x_t=1) THEN BEGIN accum=taccum;dlreg=tdlreg;
                              v:=B[2*A[v]-A[v0]] +2^f; END
                   ELSE v:=v0;
     END;       {Adding A[v0] to the accumulator (or not) and computing v}
```

5.2.3. *Termination.* No operations, except e.g. closing the code file, are necessary to terminate the decoding process.

Part 3

Measurements and Conclusion

CHAPTER 7

Measurements

1. The Compression of the CTW-2 Algorithm

bib	111261	characters
book1	768771	characters
book2	610856	characters
geo	102400	bytes
news	377109	characters
obj1	21504	bytes
obj2	246814	bytes
paper1	53161	characters
paper2	82199	characters
paper3	46526	characters
paper4	13286	characters
paper5	11954	characters
paper6	38105	characters
pic	513216	bytes
progc	39611	characters
progl	71646	characters
progp	49379	characters
trans	93695	characters

TABLE 7.1. The Calgary corpus

1.1. The data set : Calgary Corpus. To evaluate the practical performance of our implementation of the CTW algorithm, just like for the CTW-1 project, we use the Calgary corpus. This corpus is a collection of nine different types of text and to confirm that the performance of algorithms is consistent for any type, many of the types have more than one representative. Normal English, both fiction and non-fiction, is represented by two books and six papers. More unusual styles of English writing are found in a bibliography (bib) and a batch of unedited news articles (news). Three computer programs represent artificial languages (progc, progl, progp). Also a transcript of a terminal session (trans) is included. All the files mentioned so far are ASCII files. Some non-ASCII files are also included: some geophysical data (geo), two files of executable programs (obj1, obj2) and a bit-map black-and-white picture (pic). All the files and their dimensions are given once more in Table 7.1.

The file geo is particularly difficult to compress because it contains a wide range of data values, while the file pic is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros. The initial part of the file obj2 is not representative of the remainder of the file.

1.2. The standard and reference parameter sets. All individual files of the corpus were processed with the *standard* set of parameters, i.e. tree-depth $D = 6$ bytes, the zero-redundancy estimator is used. The a's and b's are counted in $k = 8$ bits (this "implies" that all probabilities are represented in logarithmic form with $m = 7$ bits fractional part, the log-table contains 256 entries, the Jacobian table contains 9*128=1152 entries, $\log \beta$ is bounded in absolute value by 18 (so, $B = 18 * 128 = 2304$), and both zero-redundancy tables contain 256 entries). Each record contains $2 * 8 + \lceil \log 36 \rceil + 7 = 29$ bits (4 bytes).

In the arithmetic encoder and decoder we work with $f = 12$-bit floating point numbers (this implies that we have an exp-table with 4096 entries and a log-table with 4095 entries).

There is no bound on the number of nodes that are generated by the CTW method. i.e. the hash table is so large that all nodes can be allocated.

In Tables 7.2 and 7.3 we see the influence of k, f, and B, the bound on the absolute value of β, on the compression rate. From these tables we conclude that for both the Krichevski-Trofimov estimator and the zero-redundancy estimator a slightly better choice for B would be $B = 1024$.

So, we define the *reference* set of parameters to be: $D = 6$ bytes, $k = 8$, $B = 1024$, and $f = 12$.

Krichevski-Trofimov estimator, maximal context depth $D = 6$ symbols

k=6

$\beta =$	4096	2304	2048	1024	512	256
f=10						
book2	2.017	2.016	2.016	2.011	2.001	2.021
geo	4.562	4.559	4.558	4.554	4.543	4.542
obj2	2.650	2.638	2.635	2.611	2.573	2.556
paper1	2.491	2.491	2.490	2.482	2.467	2.470
f=12						
book2	2.016	2.015	2.015	2.010	2.000	2.019
geo	4.563	4.560	4.559	4.554	4.543	4.542
obj2	2.651	2.639	2.636	2.612	2.574	2.556
paper1	2.492	2.491	2.491	2.482	2.468	2.469
f=14						
book2	2.018	2.017	2.017	2.012	2.002	2.019
geo	4.564	4.561	4.560	4.556	4.544	4.542
obj2	2.654	2.641	2.638	2.614	2.576	2.556
paper1	2.495	2.494	2.494	2.485	2.471	2.469
f=16						
book2	2.021	2.020	2.020	2.015	2.004	2.019
geo	4.565	4.562	4.561	4.557	4.546	4.542
obj2	2.657	2.644	2.641	2.617	2.579	2.557
paper1	2.498	2.498	2.497	2.489	2.474	2.470

k=7

$\beta =$	4096	2304	2048	1024	512	256
f=10						
book2	1.982	1.978	1.977	1.966	1.976	2.320
geo	4.550	4.545	4.543	4.531	4.522	4.670
obj2	2.626	2.606	2.601	2.563	2.537	2.886
paper1	2.464	2.457	2.456	2.441	2.438	2.713
f=12						
book2	1.980	1.976	1.975	1.964	1.975	2.319
geo	4.550	4.545	4.544	4.531	4.522	4.670
obj2	2.626	2.606	2.601	2.562	2.537	2.886
paper1	2.463	2.457	2.455	2.440	2.437	2.711
f=14						
book2	1.981	1.977	1.976	1.965	1.974	2.318
geo	4.551	4.546	4.544	4.532	4.522	4.670
obj2	2.627	2.607	2.602	2.563	2.537	2.886
paper1	2.464	2.458	2.456	2.441	2.437	2.711
f=16						
book2	1.982	1.978	1.977	1.966	1.974	2.318
geo	4.551	4.546	4.545	4.533	4.522	4.670
obj2	2.628	2.608	2.603	2.564	2.537	2.886
paper1	2.465	2.460	2.458	2.442	2.437	2.711

k=8

$\beta =$	4096	2304	2048	1024	512	256
f=10						
book2	1.960	1.953	1.951	1.957	2.306	3.184
geo	4.544	4.535	4.532	4.525	4.675	5.076
obj2	2.605	2.576	2.568	2.542	2.900	3.910
paper1	2.445	2.433	2.430	2.426	2.704	3.458
f=12						
book2	1.958	1.951	1.949	1.956	2.304	3.183
geo	4.544	4.535	4.532	4.525	4.675	5.076
obj2	2.605	2.575	2.568	2.542	2.900	3.910
paper1	2.443	2.432	2.429	2.425	2.703	3.457
f=14						
book2	1.958	1.951	1.949	1.956	2.304	3.182
geo	4.545	4.536	4.533	4.525	4.675	5.076
obj2	2.605	2.575	2.568	2.542	2.900	3.910
paper1	2.444	2.432	2.429	2.424	2.702	3.457
f=16						
book2	1.959	1.951	1.949	1.955	2.304	3.182
geo	4.545	4.536	4.533	4.525	4.675	5.076
obj2	2.605	2.575	2.568	2.542	2.900	3.910
paper1	2.444	2.432	2.429	2.424	2.702	3.457

TABLE 7.2. The influence of the parameters on the compression (KT-estimator).

Zero-redundancy estimator, maximal context depth $D = 6$ symbols

k=6

$\beta =$	4096	2304	2048	1024	512	256
f=10						
book2	1.929	1.928	1.928	1.923	1.913	1.911
geo	4.568	4.563	4.562	4.555	4.541	4.528
obj2	2.504	2.493	2.490	2.467	2.429	2.393
paper1	2.337	2.337	2.337	2.330	2.316	2.310
f=12						
book2	1.934	1.933	1.933	1.928	1.917	1.909
geo	4.574	4.568	4.567	4.561	4.547	4.531
obj2	2.518	2.507	2.504	2.480	2.441	2.396
paper1	2.349	2.348	2.348	2.342	2.328	2.314
f=14						
book2	1.945	1.945	1.944	1.940	1.928	1.913
geo	4.581	4.576	4.574	4.569	4.554	4.536
obj2	2.536	2.525	2.522	2.497	2.457	2.403
paper1	2.365	2.365	2.365	2.359	2.344	2.323
f=16						
book2	1.958	1.958	1.958	1.953	1.941	1.919
geo	4.589	4.584	4.582	4.577	4.563	4.541
obj2	2.556	2.545	2.542	2.516	2.474	2.410
paper1	2.383	2.383	2.383	2.376	2.362	2.332

k=7

$\beta =$	4096	2304	2048	1024	512	256
f=10						
book2	1.919	1.916	1.914	1.904	1.908	2.254
geo	4.564	4.557	4.555	4.539	4.526	4.668
obj2	2.490	2.471	2.466	2.429	2.399	2.756
paper1	2.328	2.323	2.321	2.307	2.303	2.584
f=12						
book2	1.920	1.916	1.915	1.904	1.904	2.251
geo	4.568	4.560	4.558	4.542	4.526	4.668
obj2	2.498	2.478	2.473	2.435	2.399	2.756
paper1	2.334	2.328	2.327	2.313	2.302	2.582
f=14						
book2	1.927	1.923	1.922	1.911	1.904	2.251
geo	4.572	4.564	4.562	4.546	4.527	4.668
obj2	2.509	2.489	2.484	2.444	2.400	2.756
paper1	2.344	2.339	2.337	2.322	2.305	2.582
f=16						
book2	1.935	1.932	1.931	1.919	1.906	2.250
geo	4.577	4.569	4.567	4.551	4.529	4.669
obj2	2.521	2.501	2.495	2.454	2.402	2.756
paper1	2.356	2.350	2.349	2.333	2.308	2.582

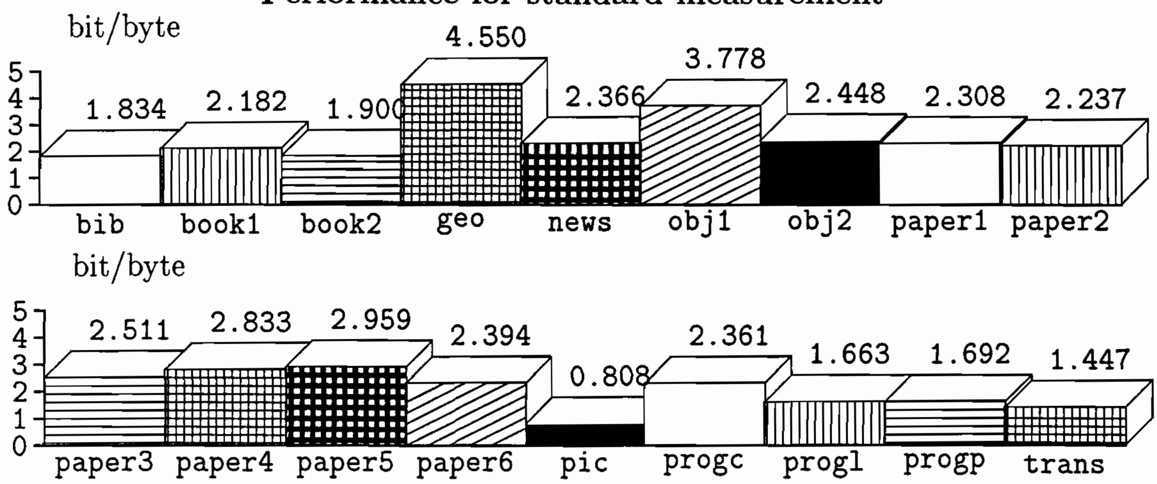
k=8

$\beta =$	4096	2304	2048	1024	512	256
f=10						
book2	1.909	1.902	1.900	1.906	2.250	3.134
geo	4.559	4.549	4.545	4.537	4.674	5.055
obj2	2.473	2.445	2.439	2.413	2.773	3.806
paper1	2.317	2.306	2.303	2.301	2.582	3.353
f=12						
book2	1.908	1.900	1.898	1.902	2.248	3.132
geo	4.561	4.550	4.547	4.537	4.674	5.055
obj2	2.476	2.448	2.441	2.412	2.773	3.806
paper1	2.319	2.308	2.305	2.299	2.580	3.351
f=14						
book2	1.912	1.904	1.902	1.901	2.247	3.132
geo	4.564	4.553	4.549	4.537	4.674	5.055
obj2	2.482	2.453	2.446	2.412	2.773	3.806
paper1	2.326	2.314	2.311	2.299	2.579	3.351
f=16						
book2	1.917	1.909	1.907	1.902	2.247	3.132
geo	4.566	4.555	4.552	4.537	4.674	5.055
obj2	2.489	2.459	2.452	2.412	2.773	3.806
paper1	2.334	2.321	2.318	2.300	2.579	3.351

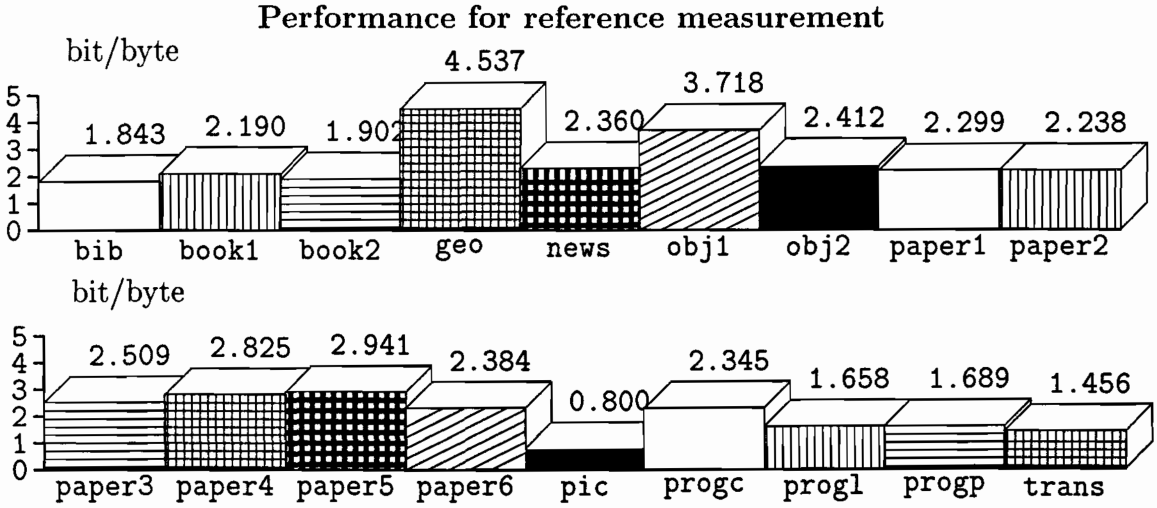
TABLE 7.3. The influence of the parameters on the compression (Zero-redundancy).

1.2.1. *COMPRESSION RATE FOR THE STANDARD AND REFERENCE PARAMETERS.* The first set of results contain the performance of the context-tree weighting method, i.e. the compression rate in bit per byte. These experiments are carried out on all files in the Calgary corpus.

Performance for standard measurement

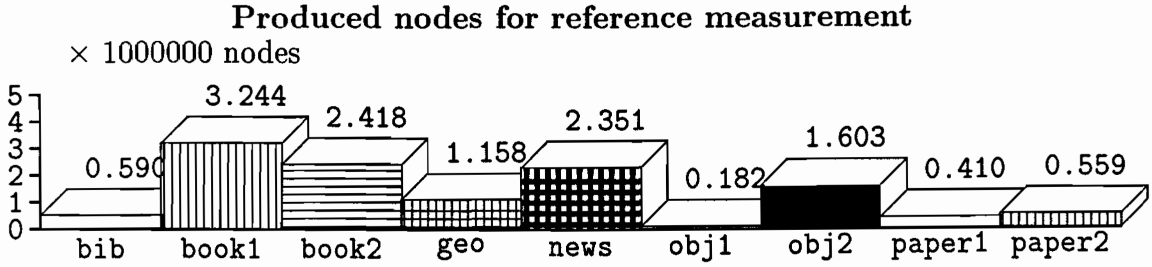


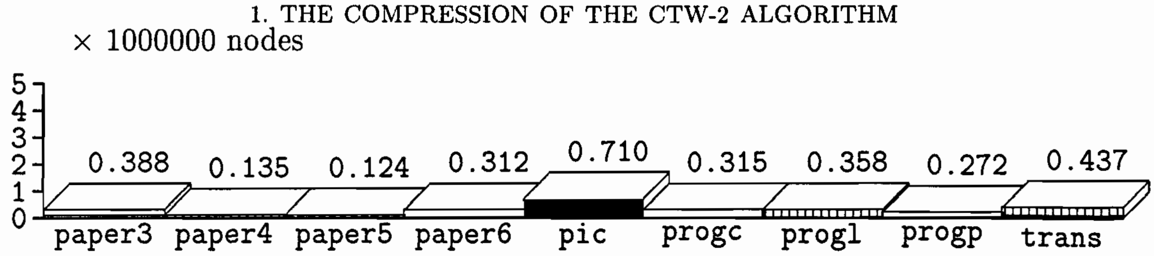
Performance for reference measurement



1.2.2. *STORAGE COMPLEXITY.*

Produced nodes for reference measurement

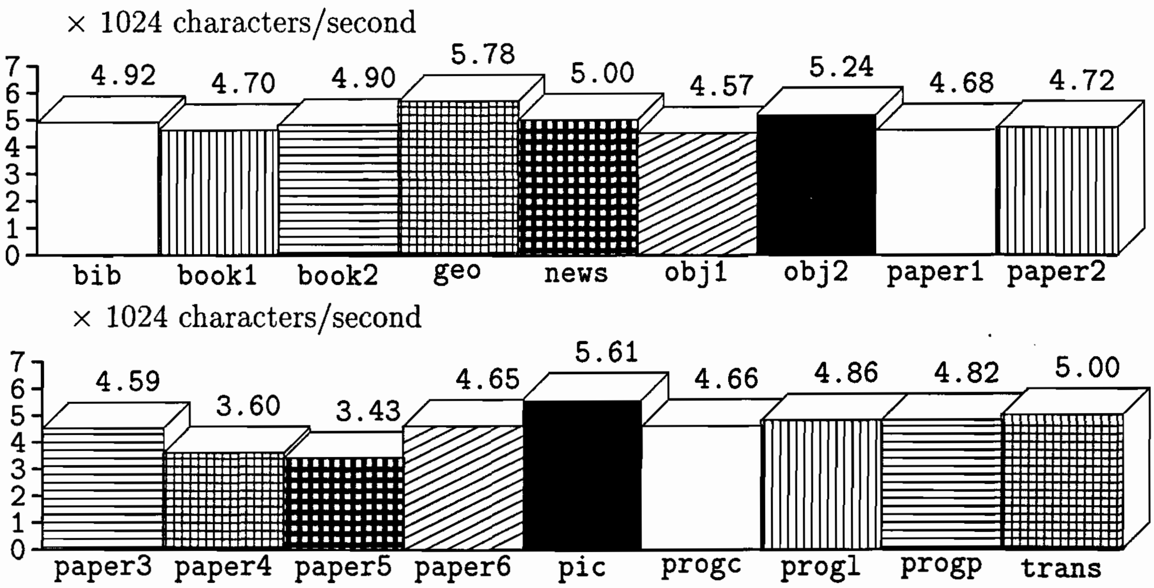




The second set of results contains the number of nodes (in multiples of 1000000 (one million)), that is produced by the context-tree weighting method for each of the files in the corpus. Note that there is no bound on this number, the context-tree can grow without limit. For large files (book1, book2, news and obj2) we therefore get a large number of nodes. Pic is also a large file but this does not produce much nodes.

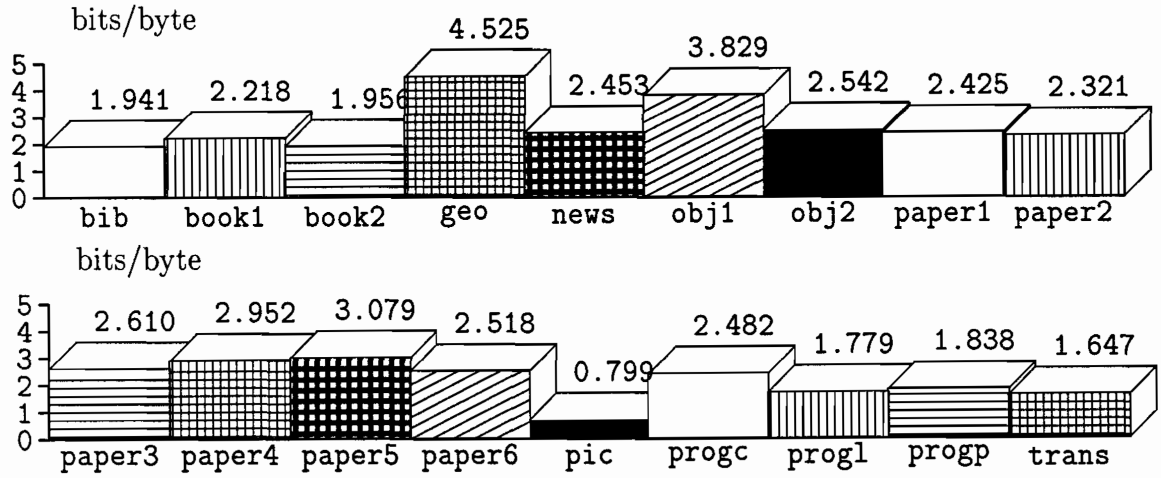
1.2.3. *DECODING SPEED.* The third set of results is the number of characters that is *decoded* per second for each of the files in the corpus. The encoding times are almost equal (within four percent of the decoding times) and are not shown here.

Decoding speed for reference measurement

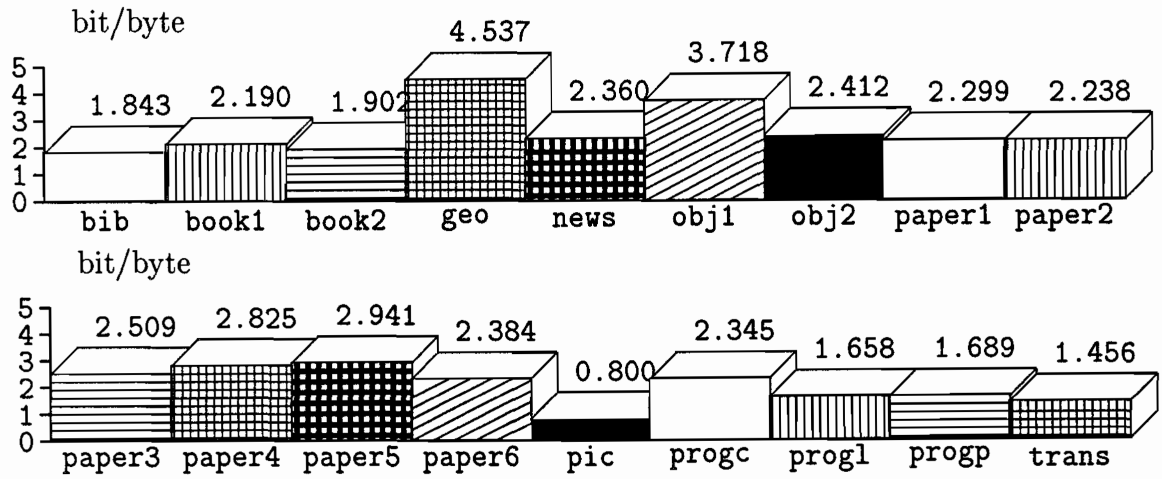


1.3. Krichevski-Trofimov versus zero-redundancy estimation measurement. The effect of using the zero-redundancy estimator is studied by comparing the results of the reference measurement to the results of an identical measurement in which the Krichevski-Trofimov estimator is used instead of the zero-redundancy estimator. So we take again depth $D = 6$ bytes, $k = 8$, $f = 12$, $B = 1024$. Again there is no bound on the number of nodes that are generated by the CTW method. The results are shown in the bar-charts below.

Performance Krichevski-Trofimov estimator

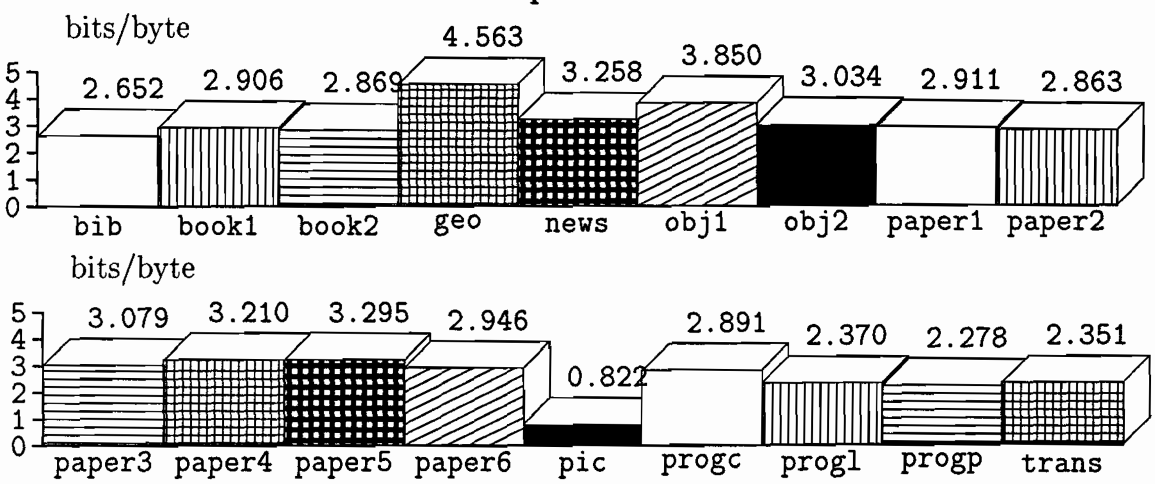


Performance zero-redundancy estimator (reference measurement)

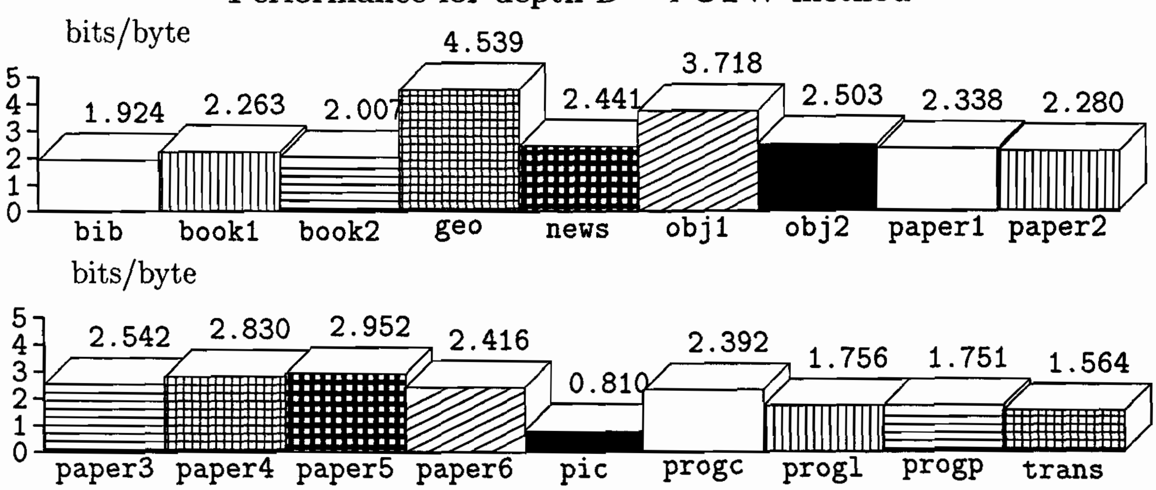


1.4. Changing the depth D of the context tree. The effect of changing the depth D of the context tree in bytes is studied in this subsection. We take as depths $D = 2, 4, 6, 8, 10$, and 12 bytes. The other parameters are determined by the reference set, i.e. $k = 8$, $B = 1024$, $f = 12$, and using the zero-redundancy estimator. There is no bound on the number of nodes that are generated by the CTW method. Note that D is the number of *complete* bytes. The most recent context part forms a incomplete byte in general and is not counted in D. The results are shown in the bar-charts below for all files in the corpus.

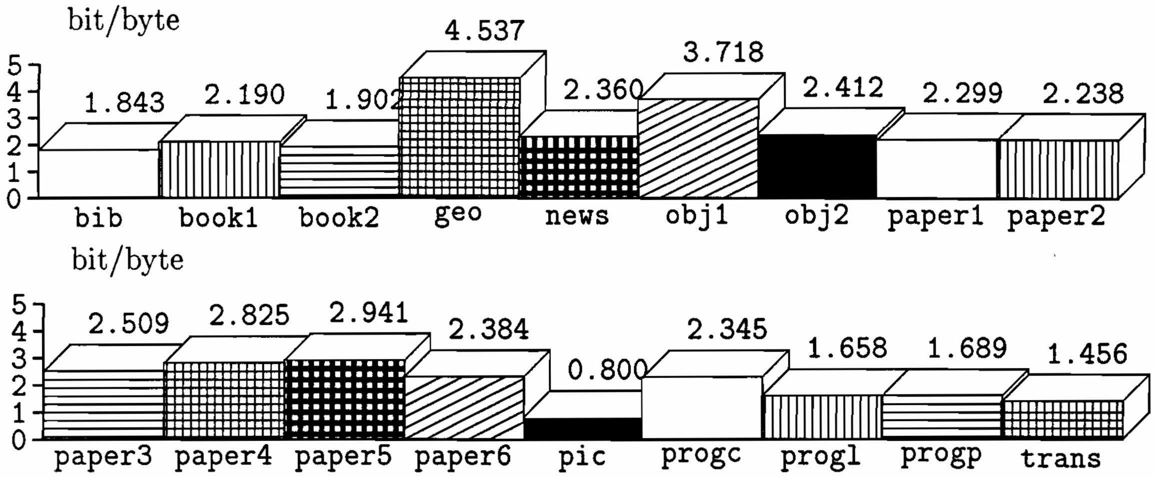
Performance for depth $D = 2$ CTW method



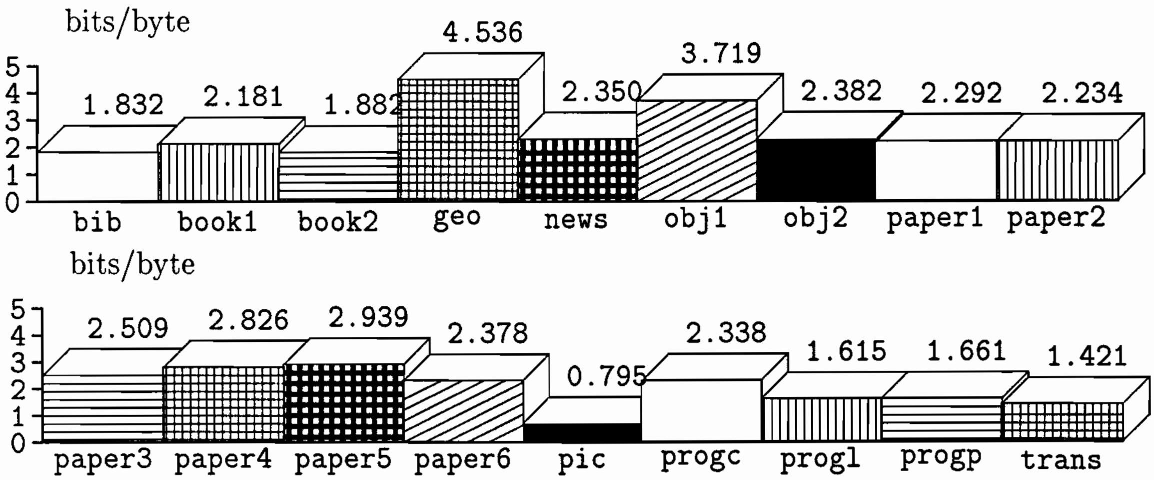
Performance for depth $D = 4$ CTW method



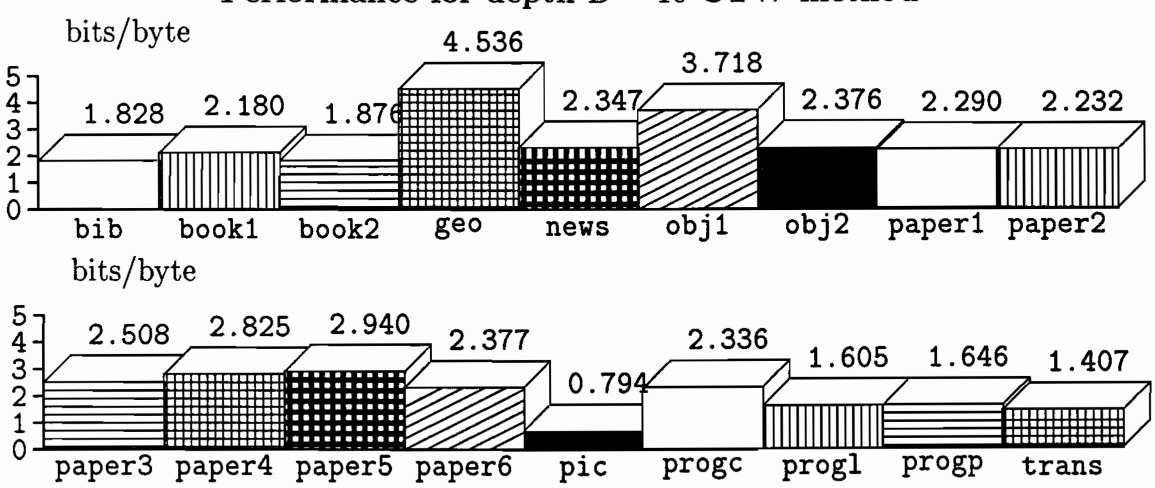
Performance for depth $D = 6$ CTW method (reference measurement)



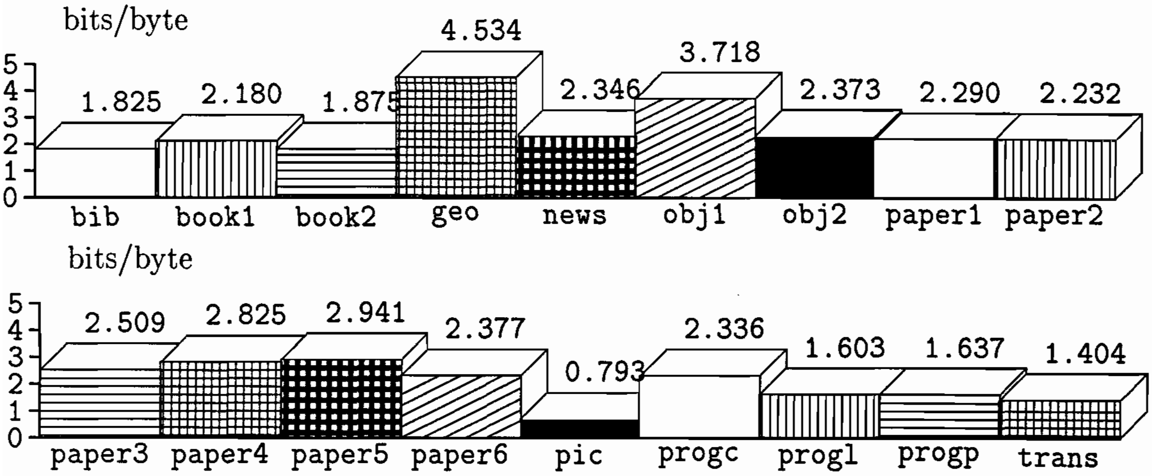
Performance for depth $D = 8$ CTW method



Performance for depth $D = 10$ CTW method



Performance for depth $D = 12$ CTW method



2. The Complexity of the CTW-2 Algorithm

In order to understand the complexity of the CTW-2 algorithm independent from the HP-9000 computer architecture we counted the different operations performed by the algorithm. We decided to categorize and count the following abstractions of operations:

incr: Increments and decrements can be special atomic operations, and so they can be quite different from ordinary additions/subtractions.

add: Additions and subtractions of 16 and 32 bit integer values.

index: Table indexing operations needed to access the tree structure and to compute the weighted probabilities.

logop: The logical operations are bitwise and and or functions, however, they implement bit masking and bit setting operations only and so can be implemented efficiently in special devices.

sft1: Shifting integer valued operands over one position to the left or right.

sftn: Shifting integer valued operands over either more than one position to the left or right or over a variable number of bit positions. This operation is usually more complex than `sft1`.

Floating point operations are only used to initialize the tables used in the weighting procedure and the arithmetic code. Although in a final implementation these tables are fixed and can be precomputed, we collected the following counts.

radd: Floating point additions.

rmult: Floating point multiplications.

rdiv: Floating point divisions. They are often more complex than multiplications and so are counted separately.

rexp: Logarithms and exponentiations on floating point values.

For each of the files in the Calgary corpus we counted the operations for the CTW-2 algorithm, with the reference parameters. In Table 7.4 we list the number of operations needed per byte for each file when using the Krichevski-Trofimov estimator or the zero-redundancy estimator, when the CTW context depth was limited to 2.

We observe that the majority of operations are additions, `add`, and table references, `index`. This is as expected, since the CTW algorithm uses simple arithmetic to compute the probabilities and tables are used to aid in this and the context data is also stored in a table.

The initialization of the tables required 2561 `radd` operations, 2561 `rmult`, 1408 `rdiv`, and 1408 `rexp` operations for the Krichevski-Trofimov estimator. In the case of the zero-redundancy estimator, initialization took 4601 `radd`, 4346 `rmult`, 2683 `rdiv`, and 1918 `rexp` operations.

In Table 7.5 we list the same for a maximal context depth of 6 bytes. The number of operations per symbol increases by a factor of 2.2. This indicates that not all contexts are continued up to depth 6, since then the amount of work should have been tripled. Namely, most of the work is done in finding the contexts and computing the weighting probabilities and this is linear in the length of the context path found in the tree. The reason for this improvement is the introduction of the unique path condition.

The table initialization is independent from the tree depth so it required the same number of operations as listed before.

Krichevski-Trofimov estimator						
filename	incr/byte	add/byte	index/byte	logop/byte	sft1/byte	sftn/byte
bib	274.244	1972.995	1142.331	173.677	24.573	72.343
book1	277.131	1892.757	1147.520	175.976	24.500	72.767
book2	278.393	1904.347	1149.159	177.072	24.529	72.790
geo	268.604	2046.323	1140.162	184.665	25.747	81.971
news	277.215	1916.902	1147.940	177.281	24.611	74.091
obj1	260.187	2156.715	1119.124	177.737	26.087	80.363
obj2	272.267	1951.253	1140.800	177.014	24.961	74.530
paper1	276.820	2034.593	1146.764	177.340	24.712	73.512
paper2	277.580	1986.985	1148.150	177.214	24.582	73.116
paper3	277.701	2027.791	1148.475	178.218	24.678	73.988
paper4	272.806	2144.374	1140.873	176.699	25.004	75.244
paper5	273.927	2169.051	1141.953	178.971	25.177	76.138
paper6	274.819	2065.240	1143.925	176.500	24.782	73.898
pic	267.402	2122.557	1132.499	163.927	24.588	66.107
progc	274.504	2062.934	1142.981	176.195	24.847	73.949
progl	273.637	2005.866	1141.133	173.085	24.583	71.676
progp	272.381	2045.059	1138.933	172.701	24.687	71.778
trans	272.071	1995.537	1138.485	172.094	24.615	71.771
Zero-redundancy estimator						
filename	incr/byte	add/byte	index/byte	logop/byte	sft1/byte	sftn/byte
bib	274.248	1582.072	1123.933	173.571	24.573	72.200
book1	277.132	1653.303	1131.751	175.933	24.500	72.710
book2	278.394	1657.652	1134.074	177.004	24.529	72.699
geo	268.607	1784.495	1135.214	184.621	25.747	81.912
news	277.219	1664.543	1135.454	177.213	24.611	73.995
obj1	260.205	1706.196	1111.604	177.287	26.087	79.756
obj2	272.273	1721.556	1132.473	176.774	24.961	74.204
paper1	276.829	1644.553	1131.437	177.130	24.712	73.228
paper2	277.586	1661.544	1132.614	177.095	24.582	72.954
paper3	277.715	1668.934	1133.727	178.092	24.678	73.810
paper4	272.835	1641.533	1124.973	176.476	25.004	74.932
paper5	273.957	1649.674	1127.578	178.638	25.177	75.683
paper6	274.836	1636.692	1128.553	176.267	24.782	73.572
pic	267.403	2041.939	1127.178	163.915	24.588	66.091
progc	274.513	1620.217	1127.534	175.908	24.847	73.562
progl	273.642	1597.450	1122.563	172.928	24.583	71.465
progp	272.391	1614.556	1121.612	172.425	24.687	71.403
trans	272.079	1576.839	1120.903	171.866	24.615	71.459

TABLE 7.4. The number of operations per source byte. $D = 2$, $k = 8$, $B = 1024$, $f = 12$. (encoder)

Krichevski-Trofimov estimator						
filename	incr/byte	add/byte	index/byte	logop/byte	sft1/byte	sftn/byte
bib	797.259	4830.922	2361.975	355.112	54.273	136.026
book1	892.152	4829.892	2519.391	399.069	56.583	137.173
book2	877.050	4822.778	2492.036	386.758	55.919	136.075
geo	462.078	3479.342	1699.624	292.296	43.857	125.804
news	790.937	4697.203	2352.349	372.439	54.269	137.414
obj1	541.380	3843.581	1827.713	286.833	44.878	125.211
obj2	731.750	4573.134	2245.315	349.973	52.321	136.579
paper1	732.152	4779.705	2255.903	351.139	53.051	137.841
paper2	783.150	4832.263	2339.836	363.977	54.424	137.516
paper3	720.427	4717.567	2232.057	351.537	52.995	137.447
paper4	621.240	4512.654	2050.838	326.919	49.932	135.699
paper5	600.539	4446.907	2008.920	321.526	49.046	135.531
paper6	709.442	4749.405	2214.972	345.905	52.334	137.670
pic	897.049	4657.454	2511.171	366.381	55.776	128.372
progc	703.605	4715.382	2202.168	341.449	52.064	137.219
progl	805.505	4902.215	2383.000	356.870	54.401	136.763
progp	779.356	4874.747	2339.004	349.548	53.652	136.962
trans	809.646	4963.047	2395.221	355.899	54.276	137.679

Zero-redundancy estimator						
filename	incr/byte	add/byte	index/byte	logop/byte	sft1/byte	sftn/byte
bib	797.262	2946.374	2317.564	354.716	54.273	135.497
book1	892.157	3361.393	2479.120	398.927	56.583	136.979
book2	877.055	3254.724	2449.018	386.530	55.919	135.766
geo	462.079	2492.694	1689.094	292.255	43.857	125.752
news	790.942	3098.594	2316.746	372.056	54.269	136.897
obj1	541.403	2533.547	1806.718	286.285	44.878	124.465
obj2	731.753	2976.889	2215.027	349.456	52.321	135.888
paper1	732.159	2903.405	2220.921	350.642	53.051	137.176
paper2	783.156	3059.984	2303.378	363.628	54.424	137.046
paper3	720.435	2927.780	2199.581	351.106	52.995	136.869
paper4	621.264	2658.549	2020.280	326.396	49.932	134.993
paper5	600.557	2603.666	1980.191	320.917	49.046	134.724
paper6	709.453	2843.925	2180.753	345.375	52.334	136.958
pic	897.050	4337.939	2498.062	366.362	55.776	128.345
progc	703.616	2794.442	2167.331	340.885	52.064	136.461
progl	805.508	2972.760	2336.973	356.424	54.401	136.169
progp	779.363	2941.717	2295.770	349.009	53.652	136.242
trans	809.649	2936.817	2349.289	355.277	54.276	136.850

TABLE 7.5. The number of operations per source byte. $D = 6$, $k = 8$, $B = 1024$, $f = 12$. (encoder)

Krichevski-Trofimov estimator					Zero-redundancy estimator				
operation	main	tree	math	ar. code	operation	main	tree	math	ar. code
incr	1.2 %	91 %	7.7 %	0.0 %	incr	1.2 %	91 %	7.7 %	0.0 %
add	0.3 %	14 %	85 %	0.3 %	add	0.6 %	23 %	76 %	0.5 %
index	0.8 %	52 %	47 %	0.1 %	index	0.8 %	53 %	47 %	0.1 %
logop	4.6 %	94 %	0.0 %	1.7 %	logop	4.6 %	94 %	0.0 %	1.5 %
sft1	0.0 %	99 %	0.6 %	0.0 %	sft1	0.0 %	99 %	0.6 %	0.0 %
sftn	5.8 %	83 %	5.8 %	5.4 %	sftn	5.8 %	83 %	5.8 %	4.9 %

TABLE 7.6. The distribution of operations over the various parts of the CTW algorithm. File=paper1, $D = 6$, $k = 8$, $B = 1024$, $f = 12$. (encoder)

In order to obtain more insight in the complexity of the different parts of the CTW algorithm, we also counted the operations for the four separate modules. In Table 7.6 we list the relative number of operations of the different types needed for the compression of the file `paper1` with the CTW algorithm and the reference parameters. We consider the following four parts.

main: This is the computational overhead that arises from the need to split a symbol into eight bits, to recombine it again and to store it in the buffer. The contribution of this overhead is very small as we can see in Table 7.6.

tree: This part represents the processing of the tree structure. Here we obtain and update the contexts along the current path. Thus we expect to find `index` operations along with `logop` operations in the manipulation of data and addresses.

math: The CTW probability weighting operations are concentrated in this unit. So, here we expect the majority of arithmetic operations, `add`, and also, because we use tables in the computations, we should see a significant amount of `index` operations.

ar. code: The last unit is the arithmetic code. Here we expect all kinds of operations, because the arithmetic code computes the codeword with the aid of tables we again expect `add` and `index` operations, but also because the codeword is binary we expect bit shift operations, `sftn`.

3. Comparison of the Performance of CTW-2 versus LZ Methods and CTW-1

3.1. Introduction. In the CTW-1 project we compared the CTW-1 algorithm with several versions of the Lempel-Ziv (LZ) methods in terms of compression-rate as a function of the memory requirements. In these comparisons we found that the LZ methods "LZSS" and "V42-bis" were often the best performers among the LZ methods. Here we compare the CTW-2 algorithm with these LZ methods and the CTW-1 algorithm.

The (buffer-method) LZSS requires $9n_w + 2^{14}$ bytes, where n_w is the buffer size and 2^{14} is the amount of bytes used for hashing.

The V42-bis compressor and decompressor was supplied by KPN. V42-bis is an efficient LZ-77 based scheme. Only for buffer sizes 2048 and 4096 we were able to compress and decompress correctly. Thus the V42-bis program has modest storage requirements. From a quick inspection of the KPN algorithm we observe that the cost per buffer position is one `int` and one `long int` variable. With HPUX-C this amounts to 4 bytes for an `int` and `long int` as well. So the total memory requirements are $8n_w$ bytes, where n_w is the buffer size (2048 or 4096 characters).

The CTW-1 algorithm ran with depths 2, 4, or 6 bytes and the best result was selected. The other parameters for CTW-1 were a CTW-mantissa size and Rubin register size of 16 bit each, and the zero-redundancy estimator was used. Each node (record) for the context-tree weighting method contains 48 bytes. Therefore the number of allocated bytes is 48 times the number produced nodes.

The CTW-2 algorithm also ran with tree depths 2, 4, and 6 and again the best results were plotted. We again used the zero-redundancy estimator and used a counter precision of $k = 8$ bits, the "beta" parameters were limited to an absolute value of $B = 1024$, and the precision of the arithmetic code was $f = 12$ bits.

Instead of measuring the number of bytes in a linear way, we use the $\log_{10}$ of the number of allocated bytes as measure in our plots. Therefore 4 stands for 10 Kbyte, 6 for 1 Mbyte and 8 for 100 Mbyte, etc.

For each of the files book2, geo, obj2, and paper1 we have made a plot containing the trade-off line corresponding to the four variants. These four plots are shown below.

3.2. Book2 measurement. The results for book2 in Figure 7.1 show that the LZSS curve appears to saturate at a rate of roughly 3 bits/byte, i.e. more available memory does not lead to much smaller compression-rates for this method.

The V42-bis algorithm only performs at low complexities, but there it outperforms the LZSS and CTW-1 algorithm.

The saturation rates for the context-tree weighting methods are less than 2 bit/byte. CTW-2 achieves a slightly lower rate than CTW-1 at maximal complexity, and more importantly, it does so at a greatly reduced complexity. From Figure 7.1 we observe that the complexity gain of CTW-2 over CTW-1 is about an 8 times complexity reduction except at very low rates and CTW-2 even outperforms V42-bis.

3.3. Geo measurement. The results for the file geo in Figure 7.2 show that all curves saturate, and that it does not pay off to use much memory on this file, especially for the Lempel-Ziv methods. V42-bis gives a smaller compression-rate than LZSS (about 6 versus 7 bit/symbol). The context-tree weighting methods are significantly better than the Lempel-Ziv algorithms. They achieve a compression-rate roughly between 4.5 and

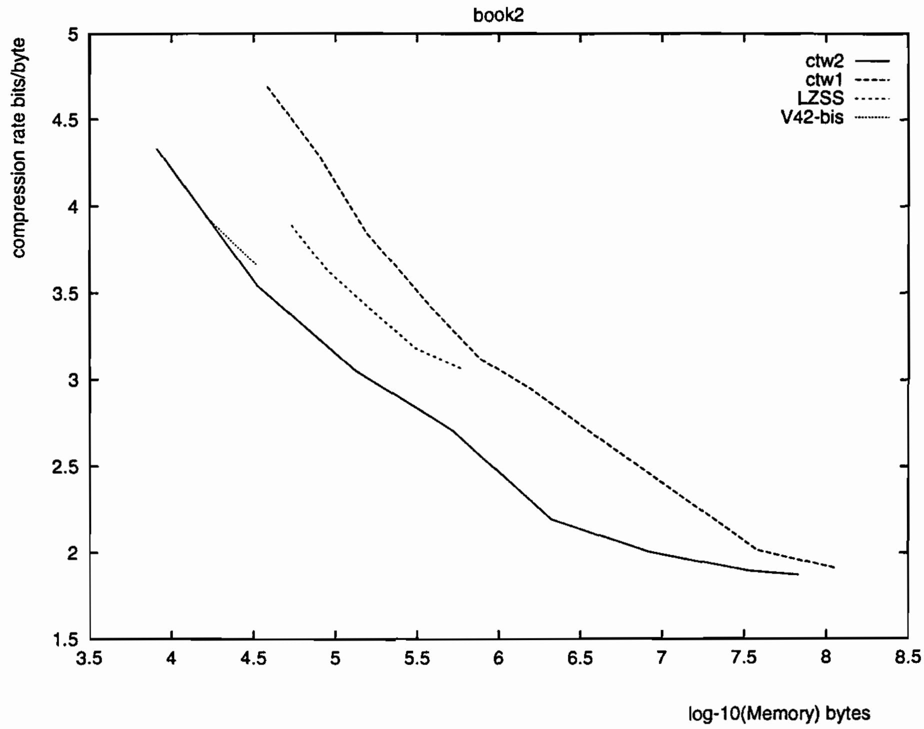


FIGURE 7.1. Memory versus compression-rate trade-off curves for book2.

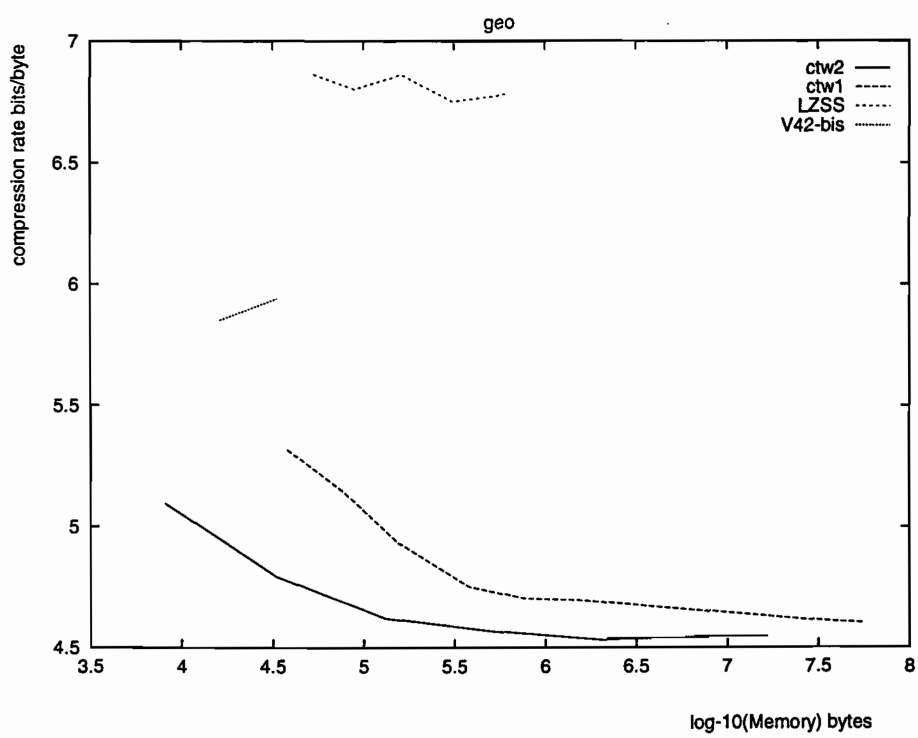


FIGURE 7.2. Memory versus compression-rate trade-off curves for geo.

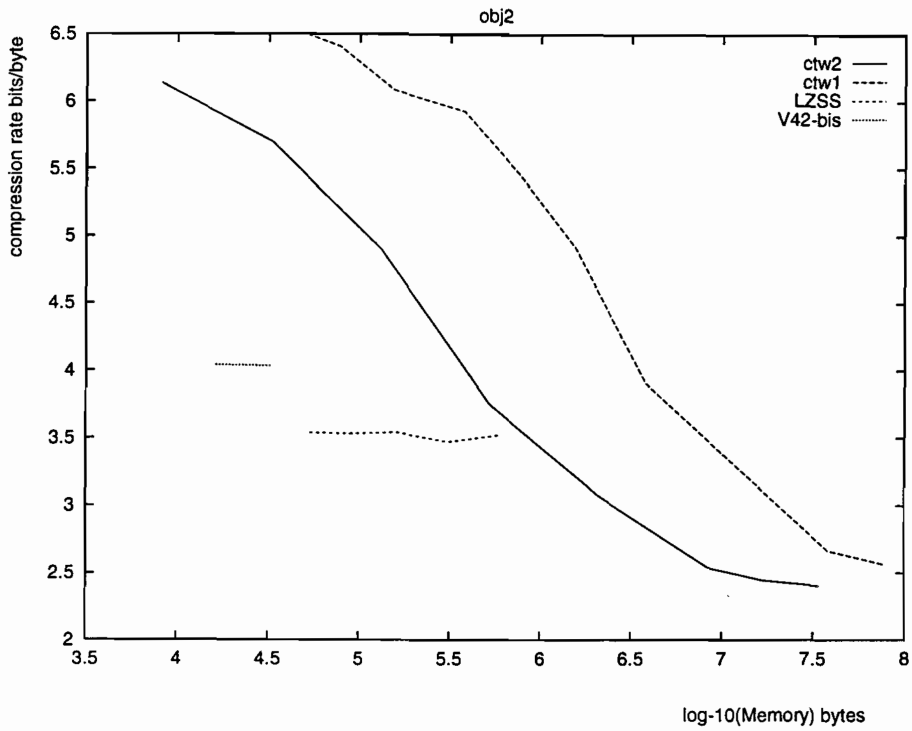


FIGURE 7.3. Memory versus compression-rate trade-off curves for obj2.

5.5 bit/byte. The CTW-2 algorithm requires much less memory than CTW-1 for a given compression rate.

3.4. Obj2 measurement. Obj2 is a file that contains a discontinuity. In the first part it contains several large blocks of repetitive data, while the second part consists of more random looking data.

It appears from Figure 7.3 that the LZSS buffer-method performs best for this file in the low complexity range. The reason for this is that the windowing effect in LZSS removes the old (non valid) statistics quickly, while the other algorithms, especially the CTW algorithms, need more conditioning (context memory depth) to separate the discontinuities. If the amount of memory is sufficiently large, even for maximum depth 2, the CTW algorithms outperform LZSS. However, this file demonstrates the fact that for repetitive data, the Lempel-Ziv methods perform very well in terms of compression rate versus storage complexity.

Again the CTW-2 algorithm reduces the storage requirements by a factor of eight or more as compared to CTW-1.

3.5. Paperl measurement. The results for the file paperl in Figure 7.4 show just like for the book2 file that the Lempel-Ziv curves saturate. LZSS and V42-bis supplement each other.

For this smaller text file the Lempel-Ziv curves saturate at rates of roughly 3.5 or 4 bit/byte. The saturation rates for the context-tree methods are roughly 2.3 bit/byte. While CTW-1 required more memory than LZSS and V42-bis in the low complexity range, CTW-2 performs even somewhat better.

The storage reduction achieved by CTW-2 relative to CTW-1 is again a factor of eight.

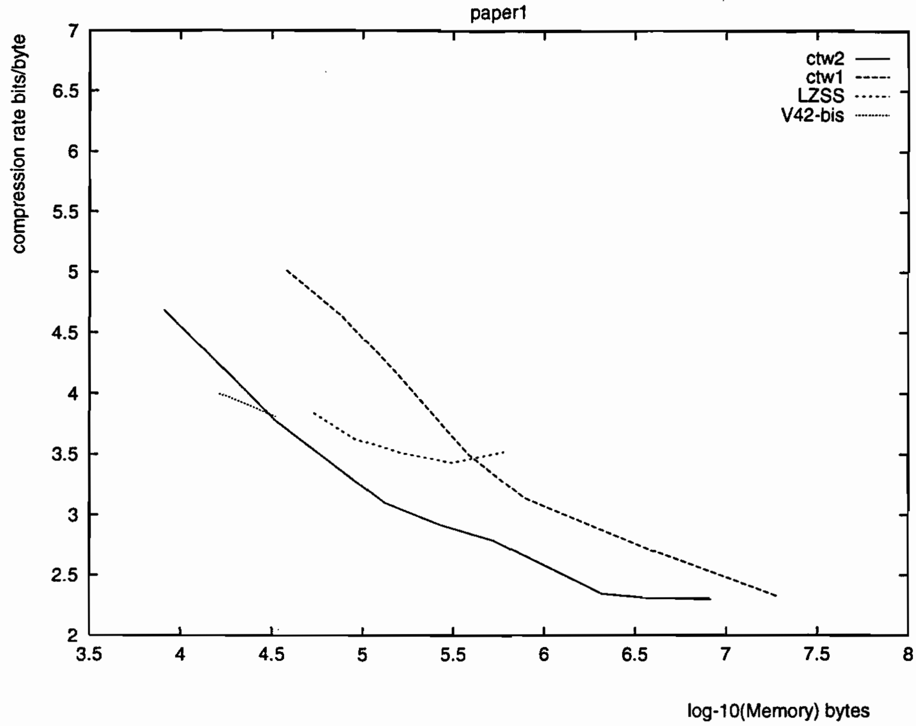


FIGURE 7.4. Memory-compression rate trade-off curves for paper1.

CHAPTER 8

Conclusion

1. Detailed Conclusions

- The (reference) context-tree weighting method achieves **compression rates** between 1.9 bit per symbol for large (book2) and 2.9 bits per symbol for small (paper5) text files. The largest rate over the corpus is achieved by the geo file (4.5 bit/symbol), the smallest rate by the file pic (0.8 bit/symbol). This is slightly better than the compression rates for CTW-1.
- The **storage complexity** for the (reference) context-tree weighting method is between 124k[1] nodes (for paper5) and 2,344k nodes (for book1). Text files achieve between 4 and 6 nodes per processed symbol for large files (bib, book1, book2 and news) and between 6 and 10 nodes per processed symbol for smaller text files (paper-files). Note that these values are identical to the results for CTW-1 as was to be expected, because the trees in CTW-1 and CTW-2 are identical. However, the **cost per node** in CTW-1 is 48 bytes, while in CTW-2 a node costs 8 bytes.
- The **decoding speed** for the (reference) context-tree weighting method is between 3.43K[2] (for paper5) and 5.78K (for geo) symbols per second. The encoding speed is roughly the same as the decoding speed. Relative to CTW-1 the speed has increased by a factor of 2.24 (paper4) to 5.25 (geo).
- The **zero-redundancy estimator** is to be preferred over the standard Krichevsky-Trofimov estimator. This holds for most of the files. Exceptions however are the files geo and pic that achieve a slightly better compression rate with the standard estimator.
- Changing the **depth** B of the context tree decreases the compression rate in general as expected. The gain is however very small for depths B larger than 6. The compression rate has improved slightly when compared to the corresponding CTW-1 results. Also, due to the smaller memory requirements, the maximal tree depth (D) considered here is 12 symbols and this improved the compression rates over the best compression rates achieved by CTW-1.
- From the tables 7.2 and 7.3 we can conclude that the **CTW maximal counter value** ($2^k - 1$) should be chosen 255, i.e. $k = 8$. The **arithmetic coding floating point size** (f) should be 12, and the **limit** (B) on the absolute value of β should then be 1024.
- Due to the better tracking of variations in the source statistics, the CTW-2 algorithm, with its limited counter range and beta range, give a slightly increased maximal compression relative to CTW-1. The increase ranges from slightly more than 1 percent for `paper1` to 6 percent for `obj2`.

[1] k=1000
[2] K=1024

- The computational load of CTW-2 is considerable. The reference algorithm needs about 6000 operations per character.
- For the whole Calgary corpus the encoding and decoding speeds of the Lempel-Ziv methods are still superior to that of the context-tree weighting algorithm CTW-2 by a factor of 30–50.
- The computational cost of the arithmetic code and the main program overhead is negligible. The table 7.6 shows that indeed the expected types of operations occur in the four units. Also, in our implementation the complexity is almost evenly spread over the `tree` and `math` module.
- Increasing the amount of memory for Lempel-Ziv algorithms above 1 Mbyte does not decrease the compression-rate. Increasing the available memory for CTW-2 decreases the achievable compression-rate (as expected). If for both classes of algorithms the amount of memory is the same and in the range from 10 Kbyte to 1 Mbyte, the compression-rate for CTW-2 is often better than the compression rate of the Lempel-Ziv algorithms. Only for `obj2` the Lempel-Ziv algorithms perform better.

2. General Conclusion

The use of the efficient integer computations, with β and the limit on its range and the count limiting procedure, improves the compression rate, reduces the memory cost, and increases the coding speed.

Also, the use of a hashing function increases the coding speed and reduces the memory cost. The CTW hashing function is simple and efficient. It employs the particularities of the context tree and thus reduces the hashing function complexity relative the the methods known from literature. This incurs an increase of less than 10 percent in the average number of probes and due to the high speed of the hashing function the overall effect is a slight increase of coding speed.

We have developed an arithmetic encoder and decoder that avoids multiplications by using exp- and log-tables. The "statistics"-input of the arithmetic encoder and decoder is the log of the conditional coding probability of the most probable symbol. Carries are avoided using a delay register. This implementation of an arithmetic code matches very well with the present CTW arithmetic.

The CTW-2 algorithm requires 8 bytes per record, while the CTW-1 algorithm needed 48 bytes. For the files we tested, i.e. `book2`, `geo`, `obj2`, and `paper1`, we found a storage reduction of a factor of eight or more. The extra gain can be explained by the improved compression rate due to the counter and "beta" limits. Thus the algorithm can now perform well in less than 32 MByte (4 million nodes) which compares favorably with the 120–190 Mbytes for CTW-1.

Bibliography

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

[2] P.E. Bender and J.K. Wolf, "New Asymptotic Bounds and Improvements on the Lempel-Ziv Data Compression Algorithm," *IEEE Trans. Inform. Theory,* vol IT-37, no 3, pp 721–729, May 1991.

[3] T.M. Cover, "Enumerative Source Encoding," *IEEE Trans. Inform. Theory,* vol. IT-19, pp. 73-77, Jan. 1973.

[4] T.M. Cover and J.A. Thomas, *Elements of Information Theory.* New York : John Wiley, 1991.

[5] I. Csiszár and J. Körner, *Information Theory, Coding Theorems for Discrete Memoryless Systems,* Budapest: Akadémiai Kiadó, 1981.

[6] L.D. Davisson, "Universal Noiseless Source Coding," *IEEE Trans. Inform. Theory,* vol. IT-19, pp. 783-795, Nov. 1973.

[7] Elias, P., "Universal Codeword Sets and Representations of the Integers," *IEEE Trans. Inform. Theory,* vol IT-21, pp 194–203, 1975.

[8] R.M. Gray and L.D. Davisson, *Random Processes : A Mathematical Approach for Engineers,"* Englewood Cliffs : Prentice-Hall, 1986.

[9] D.P. Helmbold and R.E. Schapire, "Predicting Nearly as Well as the Best Pruning of a Decision Tree," *Proc. 8th Ann. Conf. Comput. Learning Theory,* pp. 61-68, 1995.

[10] Y. Hershkovits and J. Ziv, "On Sliding-Window Universal Data Compression with Limited-Memory," submitted to *IEEE Trans. Inform. Theory.*

[11] D.A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE,* vol.40, pp. 1098-1101, Sept. 1952. Reprinted in *Key Papers in the Development of Information Theory,* (D. Slepian, Ed.) New York: IEEE Press, 1974, pp. 47-50.

[12] F. Jelinek, *Probabilistic Information Theory,* New York: McGraw-Hill, 1968, pp. 476-489.

[13] T. Kawabata, "Bayes Codes and Context Tree Weighting Method," *Trans. IEICE,* IT93-121, 1994 - 03, pp. 7-12.

[14] Knuth, D.E., *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading (Mass), 1973.

[15] R.E. Krichevsky and V.K. Trofimov, "The Performance of Universal Encoding," *IEEE Trans. Inform. Theory,* vol. IT-27, pp. 199-207, March 1981.

[16] G.G. Langdon and J. Rissanen, "Compression of Black-White Images with Binary Arithmetic Coding," *IEEE Trans. Commun.,* vol. COM-29, pp. 857-867, June 1981.

[17] T. Matsushima and S. Hirasawa, "A Universal Coding Procedure for FSMX Sources based on Bayes Coding," *Proceedings 4th Benelux-Japan Workshop on Coding and Information Theory,* Eindhoven, June 22-24, 1994, pp. 9-10.

[18] T. Matsushima and S. Hirasawa, "A Bayes Coding Algorithm Using Context Tree," *IEEE Int. Symp. Inform. Theory,* Trondheim, Norway, June 27-July 1, 1994, p. 386.

[19] T. Matsushima and S. Hirasawa, "A Bayes Coding Algorithm for FSM Sources," *IEEE Int. Symp. Inform. Theory,* Whistler, British Columbia, Canada, Sept. 17-22, 1995, p. 388.

[20] N. Merhav, "On the Minimum Description Length Principle for Sources with Piecewise Constant Parameters, *IEEE Trans. Inform. Theory,* vol. IT-39, pp. 1962-1967, November 1993.

[21] R. Pasco, *Source Coding Algorithms for Fast Data Compression,* Ph.D. thesis, Stanford University, 1976.

[22] Pearson, P.K., "Fast Hashing of Variable Length Text Strings," J. ACM., Vol. 33, No. 6, pp. 677–680, June 1990

[23] S.H.A. Peters, "Variaties op het Welch datacompressiealgorithme," Afstudeer verslag, T.U. Eindhoven, Faculteit der Elektrotechniek, oktober 1989.

[24] J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Devel.*, vol. 20, p. 198, 1976.

[25] J. Rissanen, "A Universal Data Compression System," *IEEE Inform. Theory,* vol. IT-29, pp. 656-664, September 1983.

[26] J. Rissanen, "Universal Coding, Information, Prediction, and Estimation," *IEEE Inform. Theory,* vol. IT-30, pp. 629-636, July 1984.

[27] J. Rissanen, "Complexity of Strings in the Class of Markov Sources," *IEEE Trans. Inform. Theory,* vol. IT-32, pp. 526-532, July 1986.

[28] J. Rissanen, *Stochastic Complexity in Statistical Inquiry.* Teaneck, NJ : World Scientific Publ., 1989.

[29] J. Rissanen and G.G. Langdon, Jr., "Universal Modeling and Coding," *IEEE Trans. Inform. Theory,* vol. IT-27, pp. 12-23, January 1981.

[30] F. Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers," *IEEE Trans. Inform. Theory,* vol. IT-25, pp. 672-675, November 1979.

[31] B.Ya. Ryabko, "Twice-Universal Coding," *Problems Inform. Transm.,* vol. 20, No. 3, pp. 24-28, July-September 1984.

[32] B.Ya. Ryabko, "Prediction of Random Sequences and Universal Coding," *Problems Inform. Transm.,* vol. 24, No. 2, pp. 3-14, April-June 1988.

[33] Savoy, J., "Statistical Behavior of Variable-Length Text Strings," SIGIR Forum, Vol. 24, No. 3, pp. 62–71, 1990

[34] J.P.M. Schalkwijk, "An Algorithm for Source Coding," *IEEE Trans. Inform. Theory,* vol. IT-18, pp. 395-399, May 1972.

[35] J.P.M. Schalkwijk, F.M.J. Willems, J.E. Rooijackers, and Tj.J. Tjalkens, "Results of Contract-Research at T.U. Eindhoven, A performance comparison of context-tree weighting and Lempel-Ziv algorithms," KPN Research Report (confidential), R & D-RA-95-719.

[36] C.E. Shannon, "A Mathematical Theory of Communication," *Bell Sys. Tech. Journal,* vol. 27, pp. 379-423, July, 1948. Reprinted in *Key Papers in the Development of Information Theory,* (D. Slepian, Ed.) New York: IEEE Press, 1974, pp. 5-18.

[37] Y.M. Shtarkov, "Universal Sequential Coding of Single Messages," *Problems Inform. Transm.,* vol. 23, NO. 3, pp. 3-17, July-September 1987.

[38] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, 29, no 4, 1982, pp. 928–951.

[39] J. Suzuki, "On a Generalized Context-Tree Weighting Scheme," *Proceedings 4th Benelux-Japan Workshop on Coding and Information Theory*, Eindhoven, June 22-24, 1994, p. 11.

[40] J. Suzuki, "A CTW Scheme for some FSM Models," *IEEE Int. Symp. Inform. Theory,* Whistler, British Columbia, Canada, Sept. 17-22, 1995, p. 389.

[41] J. Takeuchi and T. Kawabata, "Approximation of Bayes code for Markov Sources," *IEEE Int. Symp. Inform. Theory,* Whistler, British Columbia, Canada, Sept. 17-22, 1995, p. 391.

[42] Tj.J. Tjalkens, *Efficient and Fast Data Compression Codes for Discrete Sources with Memory.* Ph.D. dissertation, Eindhoven University of Technology, Sept. 1987.

[43] Tj.J. Tjalkens, Y.M. Shtarkov and F.M.J. Willems, "Sequential Weighting Algorithms for Multi-Alphabet Sources," *6th Joint Swedish-Russian Int. Worksh. Inform. Theory*, Mölle, Sweden, August 22-27, 1993, pp. 230-239.

[44] Tj.J. Tjalkens and F.M.J. Willems, "Arithmetic Coding," *Proc. 6th Symp. on Information Theory in the Benelux*, Mierlo, The Netherlands, May 23-24, 1985, pp. 141-150.

[45] Tj.J. Tjalkens, F.M.J. Willems and Y.M. Shtarkov, "Multi-alphabet Universal Coding Using a Binary Decomposition Context Tree Weighting Algorithm," *Proc. 15th Symp. on Information Theory in the Benelux*, Louvain-la-Neuve, May 30-31, 1994, pp. 259-265.

[46] M.J. Weinberger, A. Lempel, and J. Ziv, "A Sequential Algorithm for the Universal Coding of Finite Memory Sources," *IEEE Trans. Inform. Theory,* vol. IT-38, pp. 1002-1014, May 1992.

[47] M.J. Weinberger, N. Merhav, and M. Feder, "Optimal Sequential Probability Assignment for Individual Sequences," *IEEE Trans. Inform. Theory,* vol. IT-40, pp. 384-396, March 1994.

[48] M.J. Weinberger, J. Rissanen, M. Feder, "A Universal Finite Memory Source," *IEEE Trans. on Inform. Theory,* May 1995, pp. 643-652.

[49] T. A. Welch, "A technique for high-performance data compression," *IEEE Comput.*, vol 17, 1984, pp. 8–19.

[50] F.M.J. Willems, "Universal Data Compression and Repetition Times," *IEEE Trans. Inform. Theory*, vol. IT-35, no. 1, pp. 54–58, 1989.

[51] F.M.J. Willems, "The Context Tree Weighting Method : Finite Accuracy Effects," *Proc. 15th Symp. on Information Theory in the Benelux*, Louvain-la-Neuve, May 30-31, 1994, pp. 200-207.

[52] F.M.J. Willems, "Extensions to the Context Tree Weighting Method," Proc. IEEE 1994 Int. Symp. on Information Theory, Trondheim, Norway, June 27-July 1st, 1994, p. 387.

[53] F.M.J. Willems, "The Context-Tree Weighting Method : Extensions," accepted for publication in *IEEE Trans. on Inform. Theory.*

[54] F.M.J. Willems, "Implementing the Context-Tree Weighting Method," *Abstracts 1996 IEEE Inform. Theory Workshop*, Haifa, Israel, June 9-13, 1996, p.5.

[55] F.M.J. Willems, "Coding for a Binary Independent Piecewise Identically Distributed Source, *IEEE Trans. on Inform. Theory,* vol. IT-42, pp. 2210 - 2217, November 1996.

[56] F.M.J. Willems, Y.M. Shtarkov and Tj.J. Tjalkens, "Context Tree Weighting : A Sequential Universal Source Coding Procedure for FSMX Sources," *IEEE Int. Symp. Inform. Theory,* San Antonio, Texas, January 17-22, 1993, p. 59.

[57] F.M.J. Willems, Y.M. Shtarkov and Tj.J. Tjalkens, "The Context Tree Weighting Method : Basic properties," *IEEE Trans. on Inform. Theory,* May 1995, pp. 653-664.

[58] F.M.J. Willems, Y.M. Shtarkov and Tj.J. Tjalkens, "Context Weighting for General Finite Context Sources," to appear in *IEEE Trans. on Inform. Theory,* September 1996.

[59] Wyner, A.J., "The redundancy and distribution of the phrase lengths of the fixed-database Lempel-Ziv algorithm", in: *Proc. 1994 IEEE Int. Symp. on Inform. Theory*, Trondheim, Norway, pp. 8, June 27–July 1, 1994.

[60] Wyner, A.D. and A.J. Wyner, "Improved redundancy of a version of the Lempel-Ziv algorithm", in: *Proc. 1994 IEEE Int. Symp. on Inform. Theory*, Trondheim, Norway, pp. 9, June 27–July 1, 1994.

[61] A.D. Wyner and J. Ziv, "Some Asymptotic Properties of the Entropy of a Stationary Ergodic Data Source with Applications to Data Compression, *IEEE Trans. Inform. Theory,* pp. 1250-1258, November 1989.

[62] Wyner, A.D. and J. Ziv, "Fixed Data Base Version of the Lempel-Ziv Data Compression Algorithm," *IEEE Trans. Inform. Theory*, vol IT-37, no 3, pp 878–880, May 1991.

[63] A.D. Wyner and J. Ziv, "The Sliding-Window Lempel-Ziv Algorithm is Asymptotically Optimal, *Proc. IEEE,* June 1994, pp. 872- 877.

[64] Wyner, A.D. and J. Ziv, "The Sliding-Window Lempel-Ziv Algorithm is Asymptotically Optimal", in *Communications and Cryptography,* Ed: R.E. Blahut et al., Kluwer Acad. Publ., Dordrecht, 1994.

[65] H. Yokoo, "Improved variations relating the Ziv-Lempel and Welch-type algorithms for sequential data compression", *IEEE Trans. Inform. Theory*, vol IT-38, no 1, pp 73–81, Jan 1992.

[66] J. Ziv, "Inequalities for Source Coding : Some are More Equal Than Others," *IEEE Int. Symp. Inform. Theory,* Whistler, British Columbia, Canada, Sept. 17-22, 1995, p. 3, plenary lecture.

[67] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol IT-23, no. 3, 1977, pp. 337–343.

[68] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory*, vol IT-24, no. 5, 1978, pp. 530–536.