Weighting techniques in data compression : theory and algorithms

Weighting Techniques in Data Compression: Theory and Algorithms

PROEFONTWERP

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op vrijdag 13 december 2002 om 16.00 uur

door

Paulus Adrianus Jozef Volf

geboren te Eindhoven

De documentatie van het proefontwerp is goedgekeurd door de promotoren:

prof.ir. M.P.J. Stevens
en
prof.dr.ir. J.P.M. Schalkwijk

Copromotor:
dr.ir. F.M.J. Willems

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Volf, Paulus A.J.

Weigthing techniques in data compression : theory and algorithms / by Paulus A.J. Volf. – Eindhoven : Technische Universiteit Eindhoven, 2002.
Proefontwerp. – ISBN 90-386-1980-4
NUR 965
Trefw.: datacompressie / informatietheorie / coderingstheorie.
Subject headings: data compression / source coding / trees mathematics / information theory.

SUMMARY

THIS thesis is concerned with text compression. Text compression is a technique used on computer systems to compress data, such that the data occupies less storage space, or can be transmitted in shorter time. The compressor has to compact the data in such a way that a decompressor can restore the compacted data to its original form, without any distortion.

The algorithms discussed in this thesis process the data sequence symbol by symbol. The underlying idea of these algorithms is that the few symbols preceding the new symbol in the data sequence, called the context of that symbol, predict its probability distribution. Therefore, while processing the data sequence these algorithms collect statistics on how often a symbol follows a certain context, for different context lengths. These statistics are stored in a special data structure, the context tree. From the statistical information in the context tree the compression algorithms estimate the probability distribution of the new symbol in the data sequence for different context lengths. Since the estimates of the probability distribution for the different context lengths will differ, these estimates have to be combined into a single estimate. The main difference between the compression algorithms is in how they combine these different estimates.

The project has been set-up around the context-tree weighting (CTW) algorithm, a recently developed data compression algorithm that efficiently weights over different context lengths. The goal is to position this algorithm as a viable solution in the data compression world, next to well-known algorithms like Lempel-Ziv and the PPM-family. Therefore, the final design should be a data compression program based on CTW, that proves its applicability on a regular PC. This leads to a set of requirements for its performance, its speed and its memory usage.

The design process has led to two design proposals. The first proposed design consists of three components: an implementation of an arithmetic coder, a CTW implementation, and a forward decomposition. The arithmetic coder translates the estimated probability distributions for the data symbols into a code word. This design uses a straightforward implementation of the Witten-Neal-Cleary arithmetic coder. The CTW implementation is the modelling algorithm that estimates the probability distribution of the new symbol, based on its context, and on the data seen so far. This implementation is a refinement of a previous implementation. For the new implementation, both the computational work, and the data structures and search algorithms have been examined and redesigned. Furthermore, different pruning algorithms for the context trees have been investigated and evaluated. Finally, the parameters have been fine-tuned. The forward decomposition is a completely new component. It analyses the data in a first pass, and it converts the data symbols to bits by means of a decomposition, based on the Huffman code. The decomposition reduces the length of frequent symbols, while it tries to preserve the structure of

the original alphabet. This reduces the average number of updates of the context trees. The first complete design has been evaluated in terms of the requirements. It meets these requirements, but the performance requirement is only met in a "weak" way, i.e. only the average performance of the design, measured over a standard set of test files, is better than the performance of its competitors.

The goal of the second proposed design is to improve the performance such that it is competitive on every individual file of the standard set of test files. A new idea is proposed to achieve this: let two different modelling algorithms process the data sequence in parallel, while for each symbol a third algorithm will try to switch to the modelling algorithm that performs locally best. This idea is implemented in the second proposed design by adding two components, the snake algorithm and the companion algorithm, to the first design. The companion algorithm is the second modelling algorithm, that runs in parallel to CTW. It is based on PPMA, and it has been specifically designed to be complementary to CTW. This algorithm has been simplified, without losing its desired modelling behaviour, and it has been modified, such that it can use the same data structures as CTW. The snake algorithm is a weighting technique that for every symbol combines the two estimated probability distributions that it receives from CTW and the companion algorithm. The snake algorithm is a derivative of the more complex switching algorithm. Both algorithms have been developed for this project. They are analysed and implemented here for the first time. The complete second proposed design has been evaluated. It meets the strict definition of the performance requirement, it is one of the best data compression programs to date, at the cost of a lower speed and a higher memory usage.

By offering the user a choice between both design solutions, the project has been completed successfully.

The most important contribution of this work is the idea of switching between two modelling algorithms. One way to improve a data compression algorithm is by trying to find efficient modelling techniques for different (often larger) classes of sources. This work presents an alternative. Switching proves to be an efficient method to design a system that benefits from the strengths of both modelling algorithms, without going through the effort to capture them into a single, more complex, modelling algorithm.

SAMENVATTING

HET onderwerp van dit proefschrift is tekstcompresssie. Tekstcompressie is een techniek die op computersystemen wordt toegepast om data te comprimeren, zodat de data minder plaats in beslag nemen, of zodat deze in kortere tijd verzonden kunnen worden. De compressor moet de data wel zodanig samenpakken dat een decompressor nog in staat is de samengepakte data weer in hun oorspronkelijke vorm te herstellen, zonder dat daarbij enige mate van vervorming optreedt.

De compressie-algoritmes die in dit proefschrift besproken worden verwerken de datarij symbool voor symbool. De achterliggende gedachte bij deze algoritmes is dat het groepje symbolen dat aan het nieuwe symbool in de datarij voorafgaat, dit noemen we de context van het nieuwe symbool, zijn kansverdeling bepaalt. Daarom bouwen deze algoritmes tijdens het verwerken van de datarij statistieken op over hoe vaak een symbool op een bepaalde context volgt, en dit voor verschillende lengtes van de context. Deze statistieken worden opgeslagen in een speciale datastructuur, de context-boom. Aan de hand van de statistische informatie in de context-boom schatten de compressie-algoritmes de kansverdeling van het nieuwe symbool in de datarij, voor verschillende lengtes van de context. Aangezien de verschillende context-lengtes zullen resulteren in verschillende schattingen voor deze kansverdeling, moeten deze schattingen nog gecombineerd worden. De wijze waarop het compressie-algoritme de verschillende schattingen samenvoegt, verschilt per algoritme, en is een van de belangrijkste kenmerken van zo'n algoritme.

Het project is opgezet rond het context-boom weeg (CTW) algoritme, een recent ontwikkeld datacompressie algoritme dat op een efficiënte manier over de verschillende context-lengtes weegt. Het doel is om aan te tonen dat dit algoritme net zo bruikbaar is in praktische toepassingen als bewezen algoritmes zoals Lempel-Ziv en de PPM-familie. Daarom moet het uiteindelijke ontwerp een datacompressieprogramma worden dat gebaseerd is op het CTW algoritme, en dat op een gewone PC uitgevoerd kan worden. Om dit te realiseren zijn er eisen aan het ontwerp gesteld op het gebied van de compressieratio, snelheid en geheugengebruik.

Het ontwerpproces heeft uiteindelijk tot twee ontwerpvoorstellen geleid. Het eerste voorgestelde ontwerp bestaat uit drie componenten: een implementatie van een aritmetische coder, een CTW implementatie, en een voorwaartse decompositie. De aritmetische encoder vertaalt de geschatte kansverdeling van de datasymbolen in een codewoord. Het ontwerp gebruikt een eenvoudige implementatie van de Witten-Neal-Cleary aritmetische coder. Het CTW algoritme is het eigenlijke modelleringsalgoritme dat de kansverdeling van het nieuwe datasymbool schat, gebaseerd op zijn context en de datarij tot nu toe. De hier gebruikte implementatie is een verbetering van eerdere implementaties. Voor de nieuwe implementatie zijn zowel het rekenwerk, als

de datastructuren en de daarbij behorende zoekalgoritmes onderzocht en herontworpen. Bovendien zijn nog verschillende snoei-algoritmes voor de context-bomen onderzocht en geëvalueerd. Tenslotte zijn de optimale instellingen van de verschillende parameters vastgesteld. De voorwaartse decompositie is een compleet nieuwe component. Zij analyseert eerst de hele datarij, en bepaalt vervolgens een decompositie waarmee de datasymbolen in bits worden gesplitst. Deze decompositie is gebaseerd op de Huffman code en wordt zo geconstrueerd dat ze de lengte van veel voorkomende symbolen reduceert, terwijl zij de structuur van het originele alfabet probeert te behouden. Dit verkleint het gemiddeld aantal verversingen in de context-bomen. Het eerste volledige ontwerp is geëvalueerd om te bepalen in hoeverre er aan de gestelde eisen voldaan is. Het blijkt inderdaad aan de eisen te voldoen, maar de eis aan de compressieratio is slechts op een "zwakke" manier gehaald, dat wil zeggen, dat, gemeten over een bekende, vooraf vastgestelde, set van testfiles, alleen de gemiddelde compressie beter is dan die van concurrerende algoritmes.

Het doel van het tweede voorgestelde ontwerp is de compressie zodanig te verbeteren dat het op elke afzonderlijke file van de set van testfiles, minstens vergelijkbaar presteert met de concurrentie. Om dit te bereiken is een nieuw idee voorgesteld: laat twee datacompressie-algoritmes parallel de datarij verwerken, terwijl een derde algoritme tussen elk datasymbool zal proberen over te schakelen naar het compressie-algoritme dat op dat moment, lokaal, het beste comprimeert. In het tweede ontwerp is deze opzet geïmplementeerd door twee componenten, het snake-algoritme en het companion-algoritme, aan het eerste ontwerp toe te voegen. Het companion-algoritme is het tweede datacompressie-algoritme dat parallel zal werken aan CTW. Het is gebaseerd op PPMA, en het is speciaal ontworpen om zo complementair als mogelijk aan CTW te zijn. Dit algoritme is zoveel mogelijk vereenvoudigd, zonder dat het gewenste gedrag verloren is gegaan, en het is aangepast, zodanig dat dit nieuwe algoritme gebruik kan maken van de bestaande datastructuren voor CTW. Het snake-algoritme weegt de kansverdelingen die het voor elk symbool krijgt van zowel het CTW algoritme als het companion-algoritme. Het snake-algoritme is een afgeleide van het complexere switching-algoritme. Beide algoritmes zijn ontwikkeld voor dit project. Ze zijn hier voor het eerst geanalyseerd en geïmplementeerd. Het volledige tweede ontwerpvoorstel is geëvalueerd. Het voldoet aan de strengste compressie-eis, het is een van de beste bestaande datacompressie-programma's op dit moment, maar dit is ten koste gegaan van de compressiesnelheid en van het geheugengebruik.

Door de gebruiker een keus te bieden tussen beide oplossingen, is het project succesvol afgerond.

De belangrijkste bijdrage van dit werk is het idee om tussen twee parallel werkende modelleringsalgoritmes te schakelen. Een bekende manier om datacompressie-algoritmes te verbeteren is door te proberen steeds efficiëntere modelleringsalgoritmes te vinden, voor verschillende (vaak uitgebreidere) klassen van bronnen. Dit werk presenteert een alternatief. Het schakelen is een efficiënte methode om een systeem te ontwikkelen dat succesvol gebruik kan maken van de voordelen van twee modelleringsalgoritmes, zonder dat er veel inspanning moet worden verricht om beide algoritmes te vervangen door een enkel complex modelleringsalgoritme.

CONTENTS

1

INTRODUCTION

1.1 THE CONTEXT OF THE PROJECT

THIS thesis describes the design and implementation of a text compression technique. Text compression – the term should not be taken too literally – is a small but very lively corner of the data compression world: it is the field of the *universal lossless data compression techniques* used to compress a type of data typically found on *computer systems*.

Data compression techniques comprise all schemes that represent information in an other, more compact, form. This includes for example the Morse-code which represents the letters of the alphabet in sequences of dots, bars and spaces. One characteristic of data compression is that the compressed information cannot be used directly again: a decompression technique is required to restore the information in its original form. For example, in the old days, a telegraph operator would translate your telegram into a sequence of Morse-code symbols. These symbols are very convenient for the operator while transmitting the message over the telegraph wire, but they are more difficult to read than regular text. Therefore, the operator at the other end of the line would translate the received Morse-code symbols back into the original message.

On *computer systems* all information, or data, is stored as sequences of zeros and ones, regardless of whether the data represents an English text, a complete movie, or anything in between. On such systems, a data compression program tries to compact the data in some special format, such that the data occupies less disk space or that it can be transmitted in less time. A decompression program restores the information again.

Text compression requires that the combination of compression and decompression is *lossless*: the data after decompression should be identical to the data before compression. Lossless data compression is essential for all applications in which absolutely no distortion due to the compaction process is allowed. This is clearly necessary for source codes, executables, and texts, among others. But also for medical X-rays and audio recordings in professional studios,

lossless compression is being used. Another type of data compression is *lossy* compression, where a small amount of distortion is allowed between the decompressed data and the original data. This type of compression is almost solely used for information dealing with human perception: the human eye and ear are not perfect, they are insensitive for some specific types of distortion. This can be exploited by allowing small distortions, which are invisible, or inaudible, in the decompressed data. This has the advantage that very high compression rates can be achieved. Examples are the JPEG standard for still picture compression (photos), and the MPEG standards for video and audio compression. MPEG compression is used on DVD-systems, and for broadcasting digital TV-signals.

Finally, text compression techniques have to be *universal*. This means that they should be able to process all types of data found on computer systems. It is impossible for a text compression program to compress all possible computer files, but at least it should be able to compress a wide range of different types of files. This will be explored further in Section 1.4.

In short, the goal of this project is to develop and implement a simple utility on a computer system that can compress and decompress a file. The design should achieve the best possible compression ratio, with the limited resources of a present-day PC. As a result, there are strict constraints on the memory usage and the compression speed of the design. One of the motivations of this project is the wish to show that the Context-Tree Weighting (CTW) algorithm [62], a relatively new data compression technique, is as useful in practical applications as some other, more mature algorithms, like Lempel-Ziv and PPM. Therefore, the text compression program should be based on the CTW algorithm. This is not a rigid restriction, since recently the CTW algorithm has already been used to achieve very high compression rates [48, 64].

1.2 APPLICATIONS OF DATA COMPRESSION ALGORITHMS

Universal lossless data compression techniques have many applications. The two most straightforward applications are file compression utilities and archivers. Our final design will be a *file compressor*: a utility that uses a data compression technique to compress a single computer file. An *archiver* is a utility that can compress a set of computer files into one archive. File compressors and archivers are very similar, but an archiver also has to deal with various administrative tasks, e.g., store the exact name of the file, its date of creation, the date of its last modification, etc., and often it will also preserve the directory structure. In general an archiver will compress each file individually, but some archivers can cleverly combine similar files in order to achieve some extra compression performance. Both applications require a data compression technique that delivers a good compression performance, while they also need to be available on many different platforms. Well-known file compressors are GZIP and compress, well-known archivers are ZIP, ZOO, and ARJ.

Universal lossless data compression techniques are also used by disk compressors, inside back-up devices and inside modems. In these three applications the data compression technique is transparent for the user. A *disk compressor*, e.g. Stacker, is a computer program that automatically compresses the data that is written to a disk, and decompresses it when it is being read from the disk. The main emphasis of such programs is the compression and decompression speed.

Back-up devices, like tape-streamers, often use a data compression technique to "increase" the capacity of the back-up medium. These devices are mostly used to store unused data, or to make a back-up of the data on a computer system, just in case an incident happens. Therefore, the stored data will in general not be retrieved often, and as a result the compression speed is more important than the decompression speed, which allows asymmetrical data compression techniques. Also many communication devices, like *modems*, e.g. the V42bis modem standard, and *facsimile-equipment*, have a built-in data compression program that compresses the data just before transmission and decompresses it, immediately after reception. Compression and decompression have to be performed in real time, so speed is important. An interesting issue to consider is at which point in the communication link the data is going to be compressed. It is safe to assume that data compressed once, cannot be compressed again. If the user compresses the data (either on his system with a state-of-the-art archiver, or in the communication device), the resulting data is rather compact and the telephone company has to send all data to the destination. But if the user does not compress the data, the telephone company can compress the data instead, transmit fewer bits, while still charging the user the entire connection rate.

Finally, lossless universal data compression techniques can be used as an integral part of *databases*. Simple databases will only contain text, but nowadays there are very extensive databases that contain texts in different languages, audio clips, pictures and video sequences. Data compression techniques can be used to greatly reduce the required storage space. But, the database software has to search through the data, steered by, sometimes very complex, queries, it has to retrieve a very small portion of the total amount of stored data, and finally it might have to store the modified data again. In order to be useful under such dynamic circumstances, the data compression technique will have to fulfil very strict criteria: at the very least it has to allow *random access* to the compressed data. At the moment such features can only be found in the most simple data compression techniques. An example of a complete database project with compression can be found in "Managing Gigabytes" [70].

The algorithms investigated and developed in this thesis for universal lossless data compression programs, can also be used in other applications. We illustrated this for our students with two demonstrations. The first demonstration is intended to show that compressed data is almost random. We trained a data compression algorithm by compressing several megabytes (MBytes) of English text. Next we let students enter a random sequence of zeros and ones, which we then decompressed. The resulting "text" resembled English remarkable well, occasionally even containing half sentences. The fact that compressed data is reasonably "random" is used in *cryptographic applications*, where the data is compressed before encrypting it. For the second demonstration we trained our data compression program on three Dutch texts from different authors. Then a student had to select a few sentences from a text from any of these three authors, and the data compression program had to decide who wrote this piece of text. It proved to have a success rate close to a 100 %, which was of course also due to the fact that the three authors had a completely different style of writing. This is a very simple form of *data mining*. The data compression algorithms investigated here compress the data, by trying to find some structure in the data. The algorithms can be altered such that they do not compress the data, but that they try to find the structure that describes the given data best, instead. In this way these algorithms search explicitly for rules and dependencies in the data.

1.3 AN OVERVIEW OF THIS THESIS

The structure of this thesis is as follows.

Chapter 2 gives a complete overview of the design path followed towards the final design. First the original project description is discussed, followed by a very detailed requirements specification of the final design. Then the design process is described step by step from a high level of abstraction, linking the technical chapters, Chapters 3 to 7, together. This chapter also presents some initial investigations of possible extensions to the final design, that have been considered early in the design process, but that have not been included in the final design (e.g. parallel processing, and random access). Finally, in hindsight, the design process is reviewed and some points that could have been improved are indicated.

Chapter 3 discusses in depth the implementation of an arithmetic coder for the final design. Based on the theory described in Chapter 1, two coders are investigated and evaluated on speed and performance.

Chapter 4 discusses the CTW algorithm, starting from the theory up to its final implementation. It is shown that the CTW algorithm, a data compression algorithm for binary sequences, first has to be modified to handle non-binary sources. Next, the two basic actions of CTW, searching and computing, are investigated in depth. The resulting algorithms and data structures are implemented and evaluated.

The forward decomposition, described in **Chapter 5**, is a new component. Four different decomposition strategies of the 256-ary source symbols are developed and some limitations are discussed. Finally, the implementations are evaluated.

Chapter 6 and **Chapter 7** form the main contribution of this work. A completely new system has been proposed, in which two data compression algorithms compress the source sequence simultaneously, while between source symbols a third algorithm switches to the algorithm that performs locally best. Chapter 6 describes the development, analysis and implementation of the snake algorithm, the algorithm that switches between the two data compression algorithms. Chapter 7 describes the development and implementation of the second data compression algorithm, that will run in parallel to CTW. This chapter also gives a preliminary evaluation of the complete design.

Chapter 8 consists of two parts. In the first part the integration of the five components of the final design is discussed from four different views, relevant for this design. The second half gives an elaborate evaluation of the final design. Two set-ups of the design are proposed, and they are both compared in speed and performance to other state-of-the-art data compression algorithms.

Chapter 9 concludes this thesis. It also lists the most important innovations in the final design, and it gives some recommendations for further research.

To fully understand the workings of most modern data compression algorithms, like the Context-Tree Weighting (CTW) algorithm [62], some knowledge of the underlying mathematical theory is necessary. The remainder of this chapter will briefly introduce some of the most important information theoretical concepts. But before we start this mathematical introduction, the next section will look at data compression from a more abstract point of view: what is possible, and what isn't?

1.4 SOME OBSERVATIONS

From an abstract point of view, text compression is nothing more than a simple mapping. Suppose that the original sequence, the source sequence, has length N, then a text compression program maps all binary source sequences of length N to (hopefully shorter) binary sequences, the code words. Here, we will assume that the code words form a, so-called, prefix code: each possible source sequence gets a unique code word, and a code word may not be identical to the first part (the prefix) of any other code word. In Section 1.5.2 we will show that these prefix codes are just as powerful as any other code.

The **first observation** is that the compression of some sequences inevitably results in the expansion of other sequences. For example, take all binary sequences of length 4. Suppose that all 16 possible combinations can actually occur. For some reason we decide to compress sequence 0000 into 3 bits, and give it code word 000. Since we use 000 as a code word, we cannot use code words 0000 *and* 0001, therefore only 14 possible code words of length 4 (0010 to 1111) remain. We now have 15 code words in total, one short of the 16 code words we need. This can only be solved by extending one of the other code words at the same time. Suppose we extend code word 1111 to 5 bits, then we get *two* new code words (11110 and 11111), and we have 16 code words again: 1 of 3 bits, 13 of 4 bits and 2 of 5 bits. Note that the number of sequences that we want to compress is related to the length of their code words. You cannot compress a large fraction of all possible sequences to very short code words, because the number of available short code words will run out. This concept is formalized by the Kraft-inequality (see Theorem 1.1 in Section 1.5.2). Thus a good data compression technique has to be selective.

A **second observation** is that one can achieve compression by replacing events that occur often by short code words. For example, consider the following sequence of 36 bits:

$$100011100110001011001011001000111010.$$

It might not be directly clear how this sequence can be compressed. But a data compression technique that counts pairs of bits will find that the number of occurrences of these pairs is not uniformly distributed: 00 occurs five times, 01 once, 10 eight times and 11 four times. If it replaces these pairs by code words, such that the most common pair gets the shortest code word (e.g. 00 is mapped to 10, 01 to 110, 10 to 0, and 11 to 111), then it can successfully compress this sequence (in this case to 33 bits).

10	00	11	10	01	10	00	10	11	00	10	11	00	10	00	11	10	10
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	10	111	0	110	0	10	0	111	10	0	111	10	0	10	111	0	0

Thus in this particular case, a data compression technique that counts pairs of symbols is successful in compressing the sequence. Actually, most data compression techniques use counting techniques to compress sequences.

A **third observation** is that often the data compression technique cannot apply the counting technique directly. Therefore, many data compression techniques assume that all sequences they encounter have some common type of underlying structure such that they can count events that

are important for that class of structures. This is called *modelling* and it is the most complex topic in the field of data compression. In case the compression technique should only compress e.g. English texts, then English words and the English grammar rules form the underlying structure. Within this structure the compression technique could count e.g. word occurrences and sentence constructions.

A **last observation** is that the data compression technique does not have to assume a structure that matches the structure in the sequence perfectly to be able to compress it. Our text compression program is universal, thus the data compression technique that we are going to use needs to be able to find the structure in many different types of sequences. Our technique will assume that the sequences obey a very basic structure: the probability of a source symbol is only determined by the (at most D) previous symbols in the sequence. This structure has the additional advantage that an implementation will require only simple counting mechanisms. Although many actual sequences will belong to much more complex classes of structures, often sequences from these classes will show the behaviour of the very basic structure too. So, by assuming a less specific structure, we trade in compression performance for universality and implementation complexity.

1.5 THE MATHEMATICAL FRAME WORK

This section gives a concise introduction to some key ideas from information theory.

1.5.1 SOURCES

A source generates source symbols x_n, with $n = 1, 2, \ldots, N$, each from a source alphabet $\mathcal{X}$. For now, we assume that the source alphabet is binary, thus $\mathcal{X} = \{0, 1\}$. X_n is the random variable denoting the n^{th} output of the source. A source sequence $x_1, x_2, \ldots, x_N$, also denoted by x_1^N, is generated by the source with actual probability $P_a(x_1^N) = Pr\{X_1^N = x_1^N\}$. These probabilities satisfy $P_a(x_1^N) \geq 0$, for all $x_1^N \in \mathcal{X}^N$, and

$$\sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) = 1.$$

The actual probability $P_a(x_1^N)$ is a block probability over N source symbols. It is the product of the N actual conditional probabilities of the individual source symbols, as they are assigned by the source:

$$P_a(x_1^N) = \prod_{n=1}^{N} P_a(x_n | x_1^{n-1}).$$

For the moment we will only consider memoryless sources. In that case the random variables $X_1, X_2, \ldots, X_N$ are independent and identically distributed (i.i.d.) and consequently the actual probability of a source sequence x_1^N can be computed with $P_a(x_1^N) = \prod_{n=1}^{N} P_a(x_n)$. Binary memoryless sources are defined by a single parameter θ, with $\theta = P_a(X_n = 1)$ for all n and $0 \leq \theta \leq 1$. Now, if a source sequence x_1^N generated by such a source contains t ones, then the block probability reduces to $P_a(x_1^N | \theta) = \theta^t (1 - \theta)^{N-t}$.

1.5.2 SOURCE CODES

A binary source code is a function $\mathcal{C}$ that maps all possible binary source sequences x_1^N to binary code words of variable length[1]. A sequence x_1^N corresponds to code word $c(x_1^N)$ which has length $l(x_1^N)$. If the source sequence has length $K \cdot N$, it can be split into K subsequences of length N. Each subsequence is then encoded individually with the source code, and the code words are concatenated. The decoder will be able to decode every possible series of code words back into their corresponding source sequence, if the source code is uniquely decodable. A source code is *uniquely decodable* if every possible series of code words could have been generated by only one, unique, source sequence [16].

Uniquely decodable source codes can have an undesirable property. It is possible that the decoder has to wait until the end of the series of code words, before it can start decoding the first code word. Therefore, we will restrict ourselves to a special class of source codes, called prefix codes, which do not have this property. In a *prefix code*, no code word is the prefix of any other code word. As a result, the decoder immediately recognizes a code word once it is completely received. The code words of a prefix code can be placed in a code tree, where each leaf corresponds to one code word. Such a code tree can be constructed because no code word is the prefix of any other code word.

Example: Figure 1.1 shows two source codes for source sequences of length $N = 2$. Source code C_1 is a uniquely decodable source code, but it is not a prefix code. After receiving 11 the decoder cannot decide between a concatenation of code words 1 and 1, and the first part of the code word 110 yet. Source code C_2 is a prefix code with the same code word lengths. The code tree of this prefix code is shown on the right hand side. The decoder starts in the root node λ, and for every received symbol it takes the corresponding edge, until it reaches a leaf. As soon as it finds a leaf, the code word has been received completely, and can be decoded immediately.

x_1^N	C_1	C_2
00	1	0
01	110	100
10	010	101
11	00	11

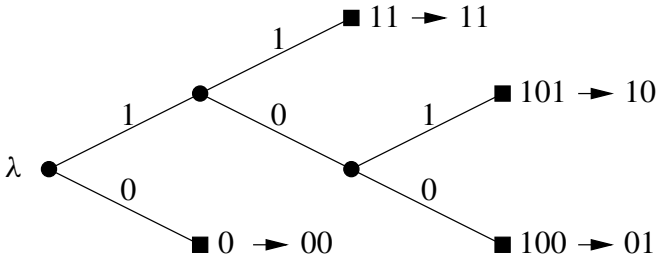


Figure 1.1: An example of a prefix code and its corresponding code tree.

An important property of any uniquely decodable code, thus also any prefix code, is that the lengths of the code words should fulfil the Kraft-inequality.

Theorem 1.1 (Kraft-inequality) *The lengths of the code words of a prefix code fulfil the Kraft-*

[1]The source code described here is a so-called fixed-to-variable length source code. The arithmetic coder described in Section 1.5.5 is a variable-to-variable length source code. There exist also variable-to-fixed length source codes, but these are not used in this work.

inequality:

$$\sum_{x_1^N} 2^{-l(x_1^N)} \leq 1, \tag{1.1}$$

in which we sum over all source sequences x_1^N for which a code word $c(x_1^N)$ exists[2]. The converse also holds. If a set of code word lengths fulfils the Kraft-inequality, then it is possible to construct a prefix code with these lengths.

Proof: The proof is based on the relation between prefix codes and code trees. Take the longest code word length, l_{max}. A tree of depth l_{max} has $2^{l_{max}}$ leaves. A code word $c(x_1^N)$ has a leaf at depth $l(x_1^N)$ in the code tree, which eliminates $2^{l_{max}-l(x_1^N)}$ leaves at the maximum depth. Because the code words fulfil the prefix condition, two different code words will correspond to different leaves, and will eliminate disjoint sets of leaves at the maximum depth. Since all code words can be placed in one code tree, the sum of all eliminated leaves at maximum depth should not exceed the total number of available leaves. Therefore:

$$\sum_{x_1^N} 2^{l_{max}-l(x_1^N)} \leq 2^{l_{max}},$$

which results in the Kraft-inequality after dividing by $2^{l_{max}}$.

The converse also holds. Suppose a set of code word lengths fulfils the Kraft-inequality. Take a tree of depth l_{max}. Take the smallest code word length, say l, from the set and create a leaf in the tree at depth l, eliminating all of its descendants. Repeat this process for the smallest unused code word length, until all code word lengths have a corresponding leaf in the tree. The resulting tree is the code tree of a prefix code. This is a valid construction of a prefix code for three reasons. First, by constructing the shortest code words first and eliminating all their descendants in the tree, we guarantee that no code word is the prefix of any other code word. Secondly, let us denote the, say M, sorted code word lengths by $l_1, l_2, \ldots, l_M$, with l_1 the smallest code word length and l_M the largest. Suppose the first m code word lengths, with $m < M$, have already been placed in the tree. Then,

$$\sum_{i=1}^{m} 2^{-l_i} < 1,$$

because the code word lengths fulfil the Kraft-inequality and there is at least one remaining code word length, l_{m+1}. Thus there is still some remaining space in the tree. Thirdly, we have to show that this remaining space is distributed through the tree in such a way, that there is still a node at depth l_{m+1} available. The construction starts with the shortest code words, and the length of the first m code words is at most l_m. A node at depth l_m in the tree is then either eliminated by one of these first m code words, or it is unused and still has all of its descendants. Therefore, there is either no space left in the tree, or there is at least one unused node available at depth l_m. Because

[2]Note that for some sources not all source sequences can occur. These sequences have actual probability 0, and a source code does not have to assign a code word to these sequences. Strictly speaking, such sources need to be treated as a special case in some of the following formulas, e.g. by excluding them from the summations. This will unnecessarily complicate the formulas for an introductory section like this, therefore we decided to omit them here.

we already know from the second step that there is still space in the tree, and because $l_{m+1} \geq l_m$, the next code word can be constructed.□

An important extension of the Kraft-inequality is given by McMillan.

Theorem 1.2 (McMillan) *The lengths of the code words of any uniquely decodable source code fulfil the Kraft-inequality.*

A proof of this theorem can be found in [16, section 5.5]. It states that the code word lengths of any uniquely decodable code must satisfy the Kraft-inequality. But a prefix code can be constructed for any set of code word lengths satisfying the Kraft-inequality. Thus by restricting ourselves to prefix codes, we can achieve the same code word lengths as with any other uniquely decodable code. Consequently, we pay no compression performance penalty by our restriction.

1.5.3 PROPERTIES OF SOURCE CODES

The redundancy is a measure of the efficiency of a source code. The individual cumulative (over N symbols) redundancy of a code word $c(x_1^N)$ (with $P_a(x_1^N) > 0$) is defined as:

$$\rho(x_1^N) = l(x_1^N) - \log_2 \frac{1}{P_a(x_1^N)}, \tag{1.2}$$

in which the term $-\log_2 P_a(x_1^N)$ is called the ideal code word length of the sequence x_1^N (this name will be explained on page 10). The individual redundancy per source symbol is $\rho(x_1^N)/N$. The entropy $H(X_1^N)$ is the average ideal code word length:

$$H(X_1^N) = \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \log_2 \frac{1}{P_a(x_1^N)}, \tag{1.3}$$

in which we use the convention $0 \log_2 0 = 0$. Now we can define the average cumulative redundancy ρ over all possible source sequences of a source code:

$$\begin{aligned} \rho &= \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N)\rho(x_1^N) = \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) l(x_1^N) - \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \log_2 \frac{1}{P_a(x_1^N)} \\ &= L - H(X_1^N). \end{aligned} \tag{1.4}$$

An optimal prefix code minimizes the average cumulative redundancy ρ by minimizing the average code word length L, defined as $L = \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \cdot l(x_1^N)$. The source coding theorem gives bounds on the average code word length of the optimal prefix code.

Theorem 1.3 (Source Coding Theorem) *Consider a source with entropy $H(X_1^N)$ and a known actual probability distribution $P_a(x_1^N)$. Then the average length L of an optimal prefix code satisfies:*

$$H(X_1^N) \leq L < H(X_1^N) + 1. \tag{1.5}$$

A prefix code achieves the entropy if and only if it assigns to each source sequence x_1^N a code word $c(x_1^N)$ with length $l(x_1^N) = -\log_2 P_a(x_1^N)$.

Proof: First we prove the right hand side of (1.5). Suppose a prefix code assigns a code word with length $l(x_1^N) = \lceil \log_2 \frac{1}{P_a(x_1^N)} \rceil$ to source sequence x_1^N. These lengths fulfil the Kraft-inequality,

$$\sum_{x_1^N \in \mathcal{X}^N} 2^{-l(x_1^N)} = \sum_{x_1^N \in \mathcal{X}^N} 2^{-\lceil -\log_2 P_a(x_1^N) \rceil} \leq \sum_{x_1^N \in \mathcal{X}^N} 2^{\log_2 P_a(x_1^N)} = 1,$$

thus it is possible to construct a prefix code with these lengths (this code is called the Shannon code, see [37]). The individual redundancy of this prefix code for a sequence x_1^N is upper bounded by,

$$\rho(x_1^N) = l(x_1^N) - \log_2 \frac{1}{P_a(x_1^N)} = \lceil \log_2 \frac{1}{P_a(x_1^N)} \rceil - \log_2 \frac{1}{P_a(x_1^N)} < 1,$$

and thus the average redundancy also satisfies $\rho < 1$. Substituting this in (1.4) results in the right hand side of (1.5). Since the optimal prefix code will not perform worse than this prefix code, it achieves the right hand side of (1.5) too.

We prove the left hand side of (1.5), by proving that $\rho \geq 0$.

$$\begin{aligned}
\rho &= \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) l(x_1^N) - \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \log_2 \frac{1}{P_a(x_1^N)} \\
&= - \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \log_2 \frac{2^{-l(x_1^N)}}{P_a(x_1^N)} \\
&= \frac{-1}{\ln 2} \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \ln \frac{2^{-l(x_1^N)}}{P_a(x_1^N)} \\
&\overset{(a)}{\geq} \frac{-1}{\ln 2} \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \left(\frac{2^{-l(x_1^N)}}{P_a(x_1^N)} - 1 \right) \\
&= \frac{-1}{\ln 2} \left(\sum_{x_1^N \in \mathcal{X}^N} 2^{-l(x_1^N)} - \sum_{x_1^N \in \mathcal{X}^N} P_a(x_1^N) \right) \\
&\overset{(b)}{\geq} \frac{-1}{\ln 2} (1 - 1) = 0,
\end{aligned} \tag{1.6}$$

where inequality (a) follows from the basic inequality $\ln a \leq a - 1$ and (b) from the Kraft-inequality (1.1). Furthermore, we use the convention $0 \log_2 0 = 0$.

An optimal prefix code has a redundancy of 0, thus it has to fulfil inequalities (a) and (b) with equality. For inequality (a) this is true only if $2^{-l(x_1^N)}/P_a(x_1^N) = 1$, and thus if $l(x_1^N) = -\log_2 P_a(x_1^N)$. Furthermore, with these code word lengths inequality (b) will be satisfied with equality too.□

The source coding theorem makes three important statements. First, the optimal prefix code has to chose the code word lengths as $l(x_1^N) = \log_2 \frac{1}{P_a(x_1^N)}$, which also explains its name "ideal code word length". Secondly, as a direct consequence, the entropy is a fundamental lower bound on the average code word length of a prefix code. Thirdly, even though the ideal code word

length does not have to be an integer value, it is possible to design a prefix code that performs within one bit of the entropy.

1.5.4 UNIVERSAL SOURCE CODING FOR MEMORYLESS SOURCES

If the parameter of the memoryless source that generated the source sequence is known, then the actual probability distribution $P_a(.)$ can be used in combination with the Shannon code to achieve a redundancy of less than 1 bit per sequence. Thus theoretically, for memoryless sources with a *known* parameter the coding problem is solved. Universal source coding deals with the situation in which this parameter, and consequently the actual probability distribution, are *unknown*.

There are different types of universal source coding algorithms. This thesis follows the approach of Rissanen and Langdon [34]. They propose to strictly separate the universal source coding process into a modelling algorithm and an encoding algorithm. The *modelling algorithm* computes a probability distribution over all source sequences: it estimates the actual probability distribution. This estimate will be used as the coding distribution $P_c(.)$ by the *encoding algorithm*. The encoding algorithm is a source code, e.g. the Shannon code, that uses this coding distribution $P_c(.)$ to code the source sequence.

For example, consider a universal source code for a binary memoryless source with unknown parameter. First, we need a good source code. The Shannon code cannot be used in practice, because it requires a table with all code words and the size of this table grows exponentially with the length of the source sequence. Section 1.5.5 discusses an arithmetic coder. This is a variable-to-variable length source code which is very computational and memory efficient, while it introduces only a very small coding redundancy of at most 2 bits. This source code operates sequentially, which means that it can process the source symbols one-by-one. In Section 1.5.6, the Krichevsky-Trofimov (KT) estimator is introduced. This estimator is a good modelling algorithm for binary memoryless sources. It also works sequentially, it estimates the probability distribution of the source symbols one-by-one. Thus, one possible universal source code for binary memoryless sources can be obtained by combining these two algorithms. Such a code will have the desirable property that it can process the source sequence sequentially: first the KT-estimator estimates the probability distribution of the next symbol in the source sequence and then the arithmetic coder codes this symbol.

Note that the KT-estimator unavoidably introduces some additional redundancy. It does not know the parameter of the source that generated the source sequence, thus it has to estimate this parameter. Moreover, the decoder does not know this parameter either, so implicitly the KT-estimator has to convey information about the parameter to the decoder. This results in some parameter redundancy. An upper bound on the parameter redundancy is computed in Section 1.5.6. Since the coding redundancy is at most 2 bits, independent of the length of the sequence, and since, as we will show later, parameter redundancy grows with the length of the sequence, the performance of universal source coding algorithms depends in general almost solely on the modelling algorithm.

1.5.5 ARITHMETIC CODING

An arithmetic encoder (the idea is from Elias, see [21]) uses the conditional coding probabilities $P_c(x_n|x_1^{n-1})$ of the individual source symbols to encode the source sequence. This requires both that the coding probabilities $P_c(x_1^N)$ of all source sequences should sum up to 1, and that the probabilities should satisfy $P_c(x_1^{n-1}) = P_c(x_1^{n-1}, X_n = 0) + P_c(x_1^{n-1}, X_n = 1)$ for all n. One can partition the interval $[0, 1)$ into disjoint subintervals, such that each possible sequence x_1^N has its own interval of size $P_c(x_1^N)$ (because of the first requirement mentioned above). Then, there exists a one-to-one relation between the sequence and its interval. We denote the lower boundary point of an interval by $Q(.)$, thus the interval corresponding to sequence x_1^N is $[Q(x_1^N), Q(x_1^N) + P_c(x_1^N))$. Any point in this interval is sufficient to specify the entire interval, and thus to specify the source sequence. The idea is shown in Figure 1.2, for sequences of length 4. Suppose the source sequence is 1010, then the code word is the binary representation of a point in this interval, which is used by the decoder to retrieve the source sequence. Two issues remain to be explained: how to compute the interval corresponding to the source sequence efficiently, and how accurate does the point in this interval have to be described?

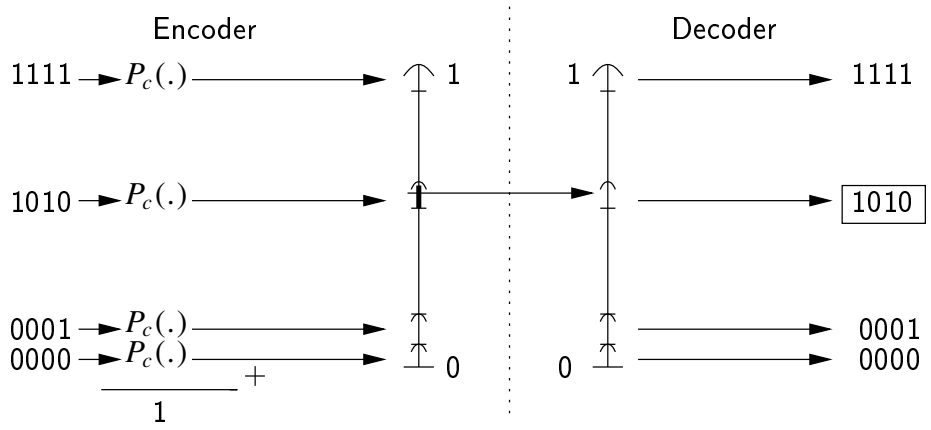


Figure 1.2: The idea behind arithmetic encoding: coding a point in an interval.

The subintervals are ordered according to the lexicographical order of the source sequences: a sequence $\tilde{x}_1^n < x_1^n$ if for some index i, $\tilde{x}_i < x_i$, and $\tilde{x}_j = x_j$ for all $j = 1, 2, \ldots, i-1$. The lower boundary point is the sum of the probabilities of all sequences with a lower lexicographical order:

$$Q(x_1^n) = \sum_{\tilde{x}_1^n < x_1^n} P_c(\tilde{x}_1^n).$$

Suppose the interval for sequence x_1^n is known and now the new interval for x_1^{n+1} has to be computed. For the lower boundary point, $Q(x_1^n 0) = Q(x_1^n)$ holds, and for the probabilities $P_c(0|x_1^n) + P_c(1|x_1^n) = 1$, thus also $P_c(x_1^n 0) + P_c(x_1^n 1) = P_c(x_1^n)$ holds (because of the second requirement mentioned at the start of this subsection). Therefore, the two subintervals for the sequences $x_1^n 0$ and $x_1^n 1$ together use exactly the same space as the interval of sequence x_1^n. Thus

extending the sequence by one source symbol, results in a split of the original interval. This gives the following very simple update rules:

$$[Q(x_1^n),\, Q(x_1^n) + P_c(x_1^n)) \xrightarrow{x_{n+1}=0} [Q(x_1^n),\, Q(x_1^n) + P_c(x_1^n 0)), \text{ and}$$
$$[Q(x_1^n),\, Q(x_1^n) + P_c(x_1^n)) \xrightarrow{x_{n+1}=1} [Q(x_1^n) + P_c(x_1^n 0),\, Q(x_1^n) + P_c(x_1^n)).$$

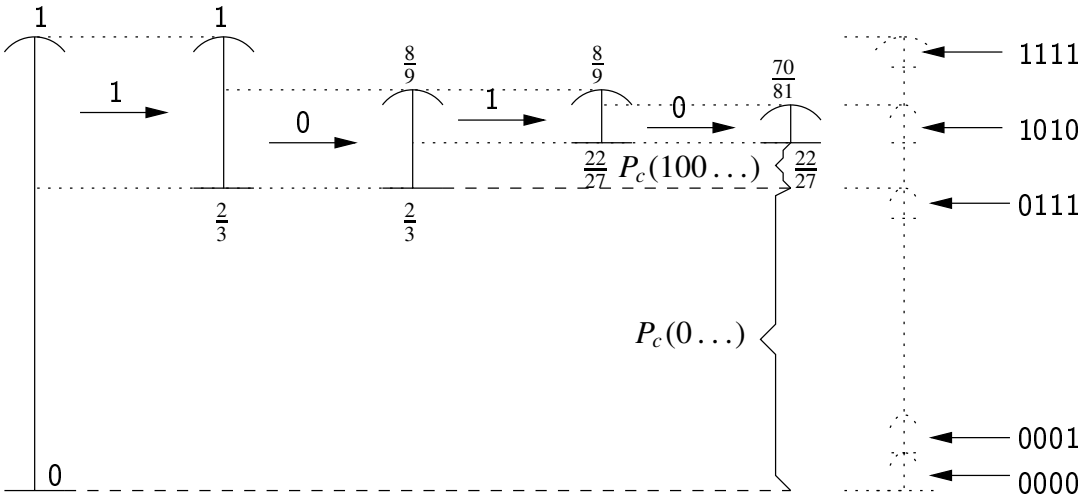


Figure 1.3: An example of the sequential idea behind arithmetic coding.

Example: Suppose the memoryless source has probability $\theta = \frac{1}{3}$ for a 1, and consider source sequence 1010... . The process of the continued interval splitting has been shown in Figure 1.3. The final size of the interval, $\frac{4}{81}$, has been found with only four multiplications, and the lower boundary point with only two additions (only if the source symbol is a 1, the probabilities have to be added). The resulting interval, $[\frac{22}{27}, \frac{70}{81})$, is exactly the same as the one which would have been found by computing the probabilities of all sequences of 4 symbols and then sorting them.

The final code word will be the binary representation of the lower boundary point, rounded up. The accuracy with which this point is sent determines the length of the code word. The rounded boundary point should be inside the interval, which has size $P_c(x_1^N)$. But recall that code words can be concatenated. If the code word has length $l(x_1^N)$, and if the next code word starts with all ones, then the received code word can be almost $2^{-l(x_1^N)}$ above the transmitted code word. Thus we have to chose $l(x_1^N)$ such that,

$$\lceil Q(x_1^N) \cdot 2^{l(x_1^N)} \rceil \cdot 2^{-l(x_1^N)} + 2^{-l(x_1^N)} \leq Q(x_1^N) + P_c(x_1^N). \tag{1.7}$$

We first upper bound the left hand part: $\lceil Q(x_1^N) \cdot 2^{l(x_1^N)} \rceil \cdot 2^{-l(x_1^N)} < Q(x_1^N) + 2^{-l(x_1^N)}$. If

$$\left(Q(x_1^N) + 2^{-l(x_1^N)}\right) + 2^{-l(x_1^N)} < Q(x_1^N) + P_c(x_1^N) \tag{1.8}$$

holds then (1.7) is satisfied too. Inequality (1.8) holds if $2^{1-l(x_1^N)} \leq P_c(x_1^N)$. Therefore the length is chosen as $l(x_1^N) = \lceil -\log_2 P_c(x_1^N) \rceil + 1$ bit. The following theorem is a direct consequence of this length assignment.

Theorem 1.4 (Coding redundancy) *The individual cumulative coding redundancy of an optimal arithmetic coder on a sequence x_1^N, given a coding distribution $P_c(x_1^N)$ is upper bounded by:*

$$\rho(x_1^N) = l(x_1^N) - \log_2 \frac{1}{P_c(x_1^N)} < 2 \quad \textit{bits.}$$

Example (continued): The source sequence 1010 has probability $\frac{4}{81}$, thus $l(1010) = \lceil -\log_2 \frac{4}{81} \rceil + 1 = 6$. Therefore, $c(1010) = \lceil Q(x_1^N) \cdot 2^{l(x_1^N)} \rceil = 53$ or 110101 in binary digits.

Arithmetic coding uses minimal resources to compute the interval. It needs to store two probabilities and the encoder uses at most two multiplications and one addition per source symbol (the decoder will be discussed in Section 3.2, but it uses a few additional operations). It constructs a code word that is less than 2 bits longer than the ideal code word length. In theory (and in practice as we will see later), the coding part is solved.

1.5.6 THE KRICHEVSKY-TROFIMOV ESTIMATOR

The Krichevsky-Trofimov estimator (KT-estimator) [26] is a modelling algorithm for binary memoryless sources. It estimates the probability distribution of the next source symbol, based on the source symbols seen so far. These estimates $P_e(.|.)$ can be used as conditional coding probabilities by the arithmetic coder. If there have been a zeros and b ones in the sequence so far, then a good estimate for the probability that the next symbol is a zero would be $\frac{a}{a+b}$. This is a good estimate, except if one of both symbols did not occur in the sequence yet, then it will get probability 0. Moreover, for the first source symbol this estimate reduces to $\frac{0}{0}$. The Krichevsky-Trofimov estimator solves this by assigning the following conditional probability to the event that symbol $X_n = 0$:

$$P_e(X_n = 0 | x_1^{n-1} \text{ contains } a \text{ zeros and } b \text{ ones}) = \frac{a + \frac{1}{2}}{a + b + 1}, \tag{1.9}$$

and similar for the probability that X_n is a 1. The first symbol now gets probability $\frac{1}{2}$. The estimated block probability of the entire source sequence x_1^N is the product of the conditional probabilities of the individual symbols. For a source sequence consisting of a zeros and b ones it reduces to:

$$P_e(x_1^N, \text{ with } a \text{ zeros and } b \text{ ones}) = P_e(a, b) = \frac{(a - \frac{1}{2})!(b - \frac{1}{2})!}{(a + b)!}, \tag{1.10}$$

where $(a - \frac{1}{2})! = \frac{1}{2} \cdot \frac{3}{2} \cdot \ldots \cdot (a - \frac{1}{2})$. The probability assigned to a sequence is independent of the order in which the source symbols occur; only the number of zeros and ones is important.

Theorem 1.5 (Parameter redundancy) *The individual cumulative parameter redundancy of the Krichevsky-Trofimov estimator on a binary sequence x_1^N generated by a memoryless source is upper bounded by:*

$$\rho_p(x_1^N) \triangleq \log_2 \frac{1}{P_e(x_1^N)} - \log_2 \frac{1}{P_a(x_1^N)} \leq \frac{1}{2}\log_2 N + 1. \tag{1.11}$$

Proof: To compute the individual redundancy of a sequence x_1^N, we need the Stirling bounds [4]:

$$\sqrt{2\pi n} n^n e^{-n} < n! < \sqrt{2\pi n} n^n e^{-n+\frac{1}{12n}}, \tag{1.12}$$

for $n = 1, 2, 3, \ldots$. First we will prove (1.11) for $a > 0$ and $b > 0$. With the Stirling bounds we can approximate the probability $P_e(a, b)$. We get,

$$P_e(a, b) > \frac{\sqrt{2}}{\sqrt{\pi} e^{\frac{1}{6}} e^{\frac{1}{24}}} \frac{1}{\sqrt{a+b}} \left(\frac{a}{a+b}\right)^a \left(\frac{b}{a+b}\right)^b, \tag{1.13}$$

in which we also used $(a - \frac{1}{2})! = \frac{(2a)!}{2^{2a} a!}$. Suppose the source has parameter θ, and the sequence has a zeros and b ones, then the actual probability $P_a(x_1^N)$ the source assigns to this sequence satisfies:

$$P_a(x_1^N) = (1-\theta)^a \theta^b \leq \left(\frac{a}{a+b}\right)^a \left(\frac{b}{a+b}\right)^b,$$

since $(1-\theta)^a \theta^b$ achieves its maximum for $\theta = \frac{b}{a+b}$. Now, the individual parameter redundancy of a sequence x_1^N satisfies,

$$\begin{aligned}
\rho_p(x_1^N) &= \log_2 \frac{1}{P_e(a, b)} - \log_2 \frac{1}{P_a(x_1^N)} = \log_2 \frac{(1-\theta)^a \theta^b}{P_e(a, b)} \\
&\overset{(a)}{\leq} \log_2 \frac{\sqrt{\pi} e^{\frac{1}{6}+\frac{1}{24}}}{\sqrt{2}} + \frac{1}{2}\log_2(a+b) + \log_2 \frac{(1-\theta)^a \theta^b}{\left(\frac{a}{a+b}\right)^a \left(\frac{b}{a+b}\right)^b} \\
&\overset{(b)}{<} \frac{1}{2}\log(a+b) + 0.62631,
\end{aligned} \tag{1.14}$$

where (a) follows from the probability approximation (1.13), and (b) follows from the upper bound on $P_a(x_1^N)$ and an approximation of the first term.

For $a = 0$ and $b > 0$ or $a > 0$ and $b = 0$ we approximate the actual probability by $P_a(x_1^N) \leq 1$. Now we find, with the Stirling bounds (1.12), for $P_e(0, N)$, with $N \geq 2$:

$$P_e(0, N) = \frac{(N - \frac{1}{2})!}{N!} = \frac{(2N)!}{2^{2N} N! N!} > \frac{e^{-\frac{2}{12N}}}{\sqrt{\pi}} \frac{1}{\sqrt{N}}. \tag{1.15}$$

Now the individual redundancy of a sequence containing only ones (similar for only zeros) satisfies,

$$\rho_p(x_1^N) \overset{(a)}{\leq} -\log_2 P_e(0, N) \overset{(b)}{\leq} \frac{1}{2}\log_2 N + 1, \tag{1.16}$$

in which (a) holds because $P_a(x_1^N) \leq 1$, and (b) holds for $N = 1$ since $P_e(0,1) = \frac{1}{2}$, and for $N \geq 2$, it follows from inequality (1.15). Combining (1.14) and (1.16) proves the theorem. □

Suppose that a sequence x_1^N generated by a memoryless source with unknown parameter has to be compressed. A universal source coding algorithm, consisting of the KT-estimator as the modelling algorithm, and the arithmetic coder as the encoding algorithm is used. Then the total redundancy, is upper bounded by:

$$\begin{aligned}
\rho(x_1^N) &= l(x_1^N) - \log_2 \frac{1}{P_a(x_1^N)} \\
&= \left(\log_2 \frac{1}{P_c(x_1^N)} - \log_2 \frac{1}{P_a(x_1^N)}\right) + \left(l(x_1^N) - \log_2 \frac{1}{P_c(x_1^N)}\right) \\
&\overset{(a)}{=} \left(\log_2 \frac{1}{P_e(x_1^N)} - \log_2 \frac{1}{P_a(x_1^N)}\right) + \left(l(x_1^N) - \log_2 \frac{1}{P_c(x_1^N)}\right) \\
&\overset{(b)}{<} \left(\frac{1}{2}\log_2 N + 1\right) + 2.
\end{aligned}$$

In (a) the estimated probability is used as coding probability and in (b) the bounds on the parameter redundancy (1.11) and the coding redundancy (1.4) are used. The parameter redundancy is introduced because the estimated probability, $P_e(x_1^N)$, is used as coding probability instead of the actual probability, $P_a(x_1^N)$.

1.5.7 RISSANEN'S CONVERSE

The average cumulative parameter redundancy of the Krichevsky-Trofimov estimator is upper bounded by $\rho_p < \frac{1}{2}\log_2 N + 1$. Rissanen [32] proved that this is the optimal result. He states that for any prefix code the measure of the subset $\mathcal{A}_\epsilon \subset [0,1]$ of parameters θ (the measure is defined as $\int_{\mathcal{A}_\epsilon} d\theta$) for which $\rho < (1-\epsilon)\frac{1}{2}\log N$ tends to 0 as $N \to \infty$ for any $\epsilon > 0$. Thus the optimal behaviour of the redundancy for a memoryless source with unknown parameter is about $\rho \approx \frac{1}{2}\log_2 N$, and consequently, the Krichevsky-Trofimov estimator performs roughly optimal.

This concludes the general introduction to some necessary concepts from the information theory. More specific concepts will be addressed in following chapters.

2

THE DESIGN PATH

This chapter provides an overview of the design path followed towards the final design. It starts with the original project description, which is used to write the more detailed requirements specification. Next, the design process is described step by step in a global way, linking the technical chapters, Chapters 3 to 7, together. The last part of this chapter presents some initial investigations of possible extensions to the final design, that have been considered early in the design process, but that have not been included in the final design.

2.1 ORIGINAL PROJECT DESCRIPTION

THE original project description, prepared in 1996, specified the following goal and requirements.

Goal: design a data compression program, based on the CTW algorithm, that compresses English text files.

The program has to fulfil the following requirements:

- Its performance (expressed in bits/symbol) should be comparable to competitive algorithms available at the time of *completion* of this project.
- It should compress 10000 symbols per second on a HP 735 workstation.
- It may not use more than 32 MBytes of memory.
- It should be based on the context-tree weighting (CTW) algorithm.

- The program should work reliable: the decompressed data should be identical to the uncompressed data.

The program may assume the following:

- The text files will be at most 1 million ASCII symbols long.

The core of this design will be the CTW algorithm [62]. Before the start of this new project the CTW algorithm has already been investigated intensively in the literature, and already two complete implementations of this algorithm existed. These two implementations were the starting point of our new design. The first implementation [48] achieves a performance, measured over a fixed set of files, that is comparable to the best competitive algorithms available at the start of our new project. But, this design is too slow and uses too much memory. The second implementation [65] has a slightly lower performance. But it is much faster and it uses considerably less memory. It actually already achieves the memory requirement set for our new design. At the start of this new project, we expected that a combination of the strengths of these two designs could result in an implementation that is close to the desired final design.

2.2 REQUIREMENTS SPECIFICATION

The original project description does not describe the requirements of the final design in detail. Therefore, in this section we will investigate the different aspects of data compression programs, and we will immediately redefine the requirements of our data compression program. If necessary we make the first design decisions.

First we will redefine the goal.

Goal: design a data compression program, based on the CTW algorithm, that compresses data files on computer systems.

We decided to extend the scope of our data compression program to all types of data commonly found on computer systems. Focusing on English texts alone limits the application of the new design too much. On the other hand, by increasing the number of different types of data sequences that should be processed, the performance of the final design on English texts will degrade.

We will arrange the different aspects of data compression programs in five groups. First we have a necessary *reliability requirement*. Next, we have *performance measures*, which are aspects of the data compression program that can be measured. The third group, the *algorithmic characteristics*, are aspects of the program that are typical for the data compression *algorithm* that is used. The fourth group are the *features*. These are optional aspects of the program: if they are available, the user notices their presence and can take advantage of the additional functionality. Finally, there are *implementation issues* which only depend on the implementation of the algorithm, and in general the user will not notice these.

NECESSARY REQUIREMENT:

1. It should work reliable.

PERFORMANCE MEASURES:

The performance measures are the only requirements that have been described in the original project description. But the description is rather global and needs to be specified in more detail.

2. **Performance:** The compression rate obtained by the program.

   Our requirement: Its performance should be competitive at the time of completion of the project. The performance of a data compression program is usually measured on a set of test files. These sets try to combine different types of files, such that they are representative of the type of files for which the program will be used. Two popular sets are the Calgary corpus [7] and the Canterbury corpus [5]. Here we will primarily use the Calgary corpus. The performance of our design will be compared to two state-of-the-art PPM-based algorithms (see [13]): the more theoretically based "state selection" algorithm [10], and the latest version of PPMZ, PPMZ-2 v0.7 [9].

   Still, this requirement is ambiguous: should the performance be comparable on the entire corpus, or on every individual file? We will distinguish two performance requirements. The **weak performance requirement** demands that the performance of CTW on the entire Calgary corpus should be better than or the same as that of the best competitive algorithm. The **strong performance requirement** has two additional demands. First, because text files are specially mentioned in the original project description, CTW should compress each of the four text files in the Calgary corpus, book1, book2, paper1, and paper2, within 0.02 bits/symbol of the best competitive algorithm. This corresponds to a maximum loss of about 1 % on the compressed size, assuming that text is compressed to 2 bits/symbol. Furthermore, it should compress the other files within 0.04 bits/symbol of the best competitive algorithm.

3. **Speed:** The number of symbols compressed and decompressed per second. Note that for some data compression algorithms these speeds can differ significantly, but that is not the case for CTW.

Discussion. In the original project description the measure speed is defined by a simple statement, while it actually is a complex notion. The speed that a certain implementation achieves depends on the algorithms that are implemented, the system on which it is executed, and the extent to which the code has been optimized for that system.

A system is a combination of hardware and software. Important hardware elements are the CPU, the memory and the communication between these two. For the CPU, not only the clock frequency, but also the architecture is important. Some architectural properties are: the instruction set (RISC/CISC), the number of parallel pipe-lines ("super-scalar"), the number of stages per pipe-line,

the number of parallel execution units, the width of the registers, the size of the primary instruction and data cache, the amount and speed of the secondary cache.

The memory is also important, since our design will use around 32 MBytes of memory, too much to fit in the cache of a micro-processor. This means that the data structures have to be stored in the main memory and transfered to the cache when needed. Therefore, also the probability of a cache-miss, the reading and writing speed of the main memory and the width of the memory bus are determining the speed.

Furthermore, the operating system also influences the compression speed. It can have other tasks running in the background simultaneously with the data compression program. In that case it has to switch continuously between tasks, which results in overhead, and most probably in repeatedly reloading the cache.

Because compression speed depends so much on the system, a "system-independent" measure would solve the problem. One possibility is to count the average number of computations needed to compress a bit. Later, we will indeed count the number of operations to compare two sets of computations for the CTW algorithm, but such a measure is not general enough, since the computational work (additions, multiplications) accounts for only part of the total compression time. A significant other part of the compression time is used for memory accesses. It is impossible to make a general statement how the time required for one memory access compares to the time for one computation. Therefore, it is not possible to fairly compare two different implementations, if one implementation has more memory accesses than the other one, while it uses fewer computations.

The focus of this thesis is directed towards finding good algorithms. We will use this machine, an HP 735, as a measurement instrument, to compare different algorithms. It is not intended to achieve the speed requirement by optimizing for this particular system, but only by improving the algorithms. It is possible that the source code of the final design will be made available to the public. In that case, optimizing for one machine is less useful. Therefore, we decided to keep the original, limited, speed requirement. The final design will be implemented in ANSI C. A standard available C-compiler will be used for compilation, and we will let it perform some system specific optimizations. In this way, we will not explicitly optimize for this single system, and our comparison are more representative for other systems. Note, that although we do not optimize our algorithms explicitly for this system, implicitly, but unavoidably, we do some system-dependent optimization. For example, during the design process some computations are replaced by table look-ups to improve the compression speed. Such a trade-off is clearly influenced by the hardware.

The ideas for a parallel CTW implementation (Section 2.4.1) are an exception. These ideas are developed with a multi-processor system in mind. The goal was to split-up the CTW algorithm into parallel tasks, such that the communication between the different tasks is minimized. It needs a detailed mapping to translate these ideas to a single super-scalar or VLIW processor.

Our requirement: It should compress at least 10000 symbols per second on the available HP 9000 Series 700 Model 735/125 workstation. This workstation has a 125 MHz PA7150 processor. It achieves a SPECint95 of 4.04 (comparable to a 150 MHz PentiumI processor) and a SPECfp95 of 4.55 (slightly faster than a 200 MHz PentiumI processor).

4. **Complexity:** The amount of memory used while compressing or decompressing. The complexity can be specified for the encoder and decoder separately.

   Our requirement: The CTW algorithm has the same memory complexity for compressing and decompressing. In principle, the program should not use more than 32 MBytes of memory, but because both competitive state-of-the-art data compression programs use considerably more than 32 MBytes of memory, e.g. PPMZ-2 uses more than 100 MBytes on some files, it might be acceptable for our design too, to use more than 32 MBytes.

5. **Universality:** Data compression programs can be specifically designed for sequences with one type of underlying structure, e.g. English texts. Specializing can have advantages in performance, speed and complexity. But, if such a program is applied to sequences with a different underlying structure, then the performance can be poor. Thus, there is a trade-off between the universality of the program and the performance, complexity and speed of the program.

   Our requirement: It should be able to process all data files that can be encountered on computer systems. As mentioned, this is an extension on the original goal.

ALGORITHMIC CHARACTERISTICS:

6. **Type of algorithm:** What type of algorithm is used: a dictionary algorithm, or a statistical algorithm?

   Our requirement: It should be based on the CTW algorithm.

7. **Type of encoding:** Statistical algorithms need a separate coding algorithm. What type of coding algorithm will be used: e.g. enumerative coding, arithmetic coding, or one based on table look-up.

   Our requirement: Not specified. But implicitly it is assumed that an arithmetic coder will be used, since it appears to be the only viable solution at the moment.

8. **Parallel computing:** The ability of the program to spread the work load during compression, decompression, or both, over multiple processors. In some algorithms it is easier to allow parallel compression than to allow parallel decompression. So this feature could, for example, influence the balance of a certain program.

   Our requirement: Not specified. Support for parallel processing is very desirable, but not mandatory.

9. **Balance:** The difference between the compression speed and the decompression speed. In a balanced system the two speeds are comparable.

   Our requirement: Not explicitly specified, but the CTW algorithm is a balanced algorithm.

10. **Number of passes:** The number of times the algorithm has to observe the complete data sequence before it has compressed that sequence. A *sequential* algorithm runs through the data sequence *once*, and compresses the symbols almost instantaneously. This is different from a one-pass algorithm, that might need to observe the entire data sequence before it starts compressing the sequence.

    Our requirement: Not specified. CTW is a sequential algorithm, and this is of course the most desirable form.

11. **Buffering:** The amount of delay the compressor or decompressor introduces while processing the data. This is closely related to the number of passes.

    Our requirement: Not specified. In sequential algorithms, buffering corresponds to the delay introduced by the algorithm between receiving an unprocessed symbol and transmitting it after processing. In one-pass or multi-pass systems such a definition is not useful. In that case, one might investigate how much of the source sequence needs to be buffered while processing the sequence.

    CTW is sequential and has a delay of only a few symbols. Preferably, the new design should introduce as little extra delay or buffering as possible.

FEATURES:

12a. **Random Access:** The ability of the decoder to decompress a part of the data, without the necessity to decompress *all* previous data.

12b. **Searching:** Is it possible to search for strings in the compressed data?

12c. **Segmentation:** The ability of the encoder to recognize different parts (segments) in the data, e.g. based on their underlying structure, and use a suitable algorithm for each segment.

    Our requirement: Support for **random access**, **searching**, or **segmentation** is very desirable, but not mandatory. The CTW algorithm in its current form does not allow any of these features.

IMPLEMENTATION ISSUES:

13. **Compatibility:** Is the program compatible across different platforms? Is it maybe compatible with other data compression programs?

    Our requirement: Not specified. Our final design should be compatible across different platforms: files compressed on one system should be decompressible on any other platform. In order to allow easy porting of the design, we will design the program as a set of ANSI C-files. Compatibility with other data compression algorithms is harder to achieve. This will be the first real CTW-oriented data compression program (the two older implementations are more intended for research purposes). Therefore, compatibility to other

programs can only be achieved by simply including their encoders and decoders. This is an unnecessary complication of our final design.

14. **Modular design:** Is the program completely integrated or is it designed as a collection of independent modules, which can be replaced by different modules, or which could be used in other designs?

    Our requirement: Not specified. The CTW algorithm already has, on a very basic level, a modular design. Partitioning the project in modules will not only ease the design process, but might also increase its use for future projects.

15. **Flexibility:** Is it possible to (efficiently) modify the program for special circumstances?

    Our requirement: Our new goal is to develop a universal data compression algorithm, thus it should be able to process any type of file. But it is of course possible to implement special facilities in the design for some specific types of data, e.g. pictures or texts.

    In principle we decided to implement a single universal data compression algorithm, without any special facilities, because otherwise a lot of time is invested to improve the performance of the final design for a small subset of files only. On the other hand, English texts are specifically mentioned in the original project description. A small research project has been conducted to include additional facilities for texts (see Section 2.4.3), but in the end it has not been used in the final design. As a compromise, we decided to define a stricter performance requirement for text files.

16. **Scalability:** Is it possible to scale the program? E.g., if we allow twice the memory will the program (or a slightly modified version) be able to use it efficiently, or not?

    Our requirement: Not specified. Every data compression algorithm has a maximum compression performance it can achieve given unlimited time and memory. Therefore, most data compression algorithms will perform better if they can use more processing power or more memory, but the increase in performance is in general not proportional to the increase in resources. The same holds for the CTW algorithm: more memory or more processing time will in general result in a (relatively small) increase in performance. Since, the speed and memory usage of our final design are restricted, we will not require a specific amount of scalability.

2.3 THE DESIGN STRATEGY

Table 2.1 summarizes all design criteria of our design. The three key design criteria, performance, speed and complexity, have been our main concern while making most design decisions and therefore the following chapters will focus most on achieving these criteria. In addition to these criteria, five other criteria have been considered at some point during the design process. Table 2.1 refers for each of these criteria to the corresponding sections in this thesis. The remaining criteria were not considered during the actual design process. Three features, parallel

Key design criteria:

2. Performance:
   Weak performance criterion:
   - at least as good on the entire Calgary corpus.

   Strong performance criterion:
   - at least as good on the entire Calgary corpus,
   - loss at most 0.02 bits/symbol on individual texts,
   - loss at most 0.04 bits/symbol on other files.

3. Speed: Compress 10000 symbols per second on a HP 735 work station.
4. Complexity: At most 32 MBytes.

Other design criteria:

6. Type of algorithm: CTW by specification. See Chapter 4.
10. Number of passes: Preferable sequential. Discussed in Chapter 5.
11. Buffering: As small as possible. Discussed in Sections 4.4.3, 7.3.4 and 8.1.3.
13. Compatibility: Compatible across platforms. Discussed in Sections 4.4.1 and 8.1.2.
14. Modularity: Desirable for future projects. Discussed in Section 8.1.5.

Investigated but rejected criteria (see Section 2.4):

8. Parallel compression or decompression.
12. Random access, searching and segmentation.
15. Flexibility.

Not considered explicitly:

1. Reliability: Essential property.
7. Type of encoding: Not specified.
5. Universality: Automatically fulfilled by design.
9. Balance: Not specified.
16. Scalability: Not required.

Table 2.1: An overview of all design criteria.

compression and decompression, flexibility and random access, searching and segmentation have been investigated, but already in the beginning of the design process it was decided that these would not be implemented. Some of the work on the three criteria has been mentioned in Section 2.4. Finally, some criteria were trivial, impossible to achieve or not specified. These have not been mentioned explicitly in the remainder of this work.

The remaining part of this section will follow the entire design process and it will show how the final design evolved.

2.3.1 STEP 1: THE BASIC DESIGN

Section 1.5.4 describes a universal source coding algorithm for memoryless sources. It consists of a modelling algorithm, the Krichevsky-Trofimov estimator, and a coding algorithm, an arithmetic coder. The final design has to use the CTW algorithm, which is a modelling algorithm for binary tree sources. The coding algorithm has not been specified, but at the moment the only viable choice is an arithmetic coder. A block diagram of a data compression program based on CTW has been shown in Figure 2.1. Here AE denotes Arithmetic Encoder, AD denotes Arithmetic Decoder, and $c(x_1^N)$ denotes the compressed data. The complexity of the CTW algorithm is much higher than the complexity of the arithmetic coder, thus it seems reasonable to focus on an efficient implementation for the CTW algorithm, without considering the interface with the arithmetic coder and then let the arithmetic coder solve possible communication problems.

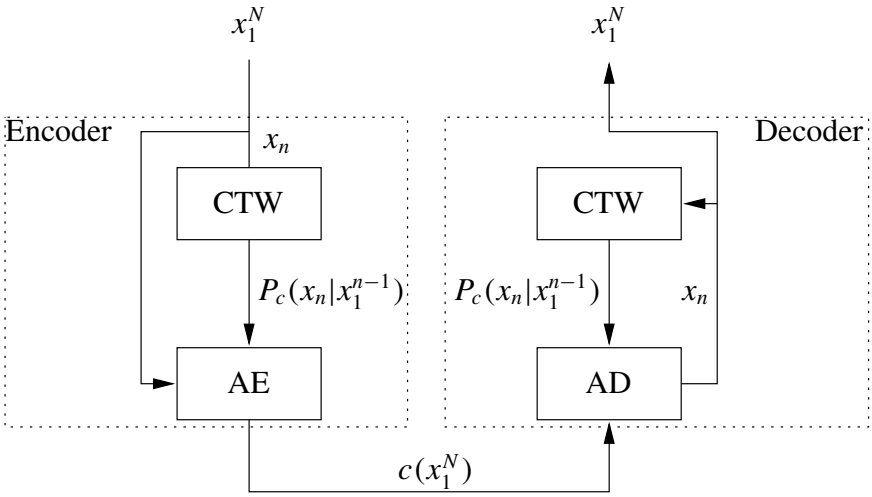


Figure 2.1: The basic CTW compressor design.

THE BASIC DESIGN: THE ARITHMETIC CODER

The principle of an arithmetic coder has been described in Section 1.5.5. The modelling algorithm, the CTW algorithm, estimates the probability distribution of the next source symbol, which is then used by the arithmetic coder to code this symbol. In our implementation the CTW

algorithm estimates the probability distribution of single bits. These estimates use a fixed-point logarithmic representation: the logarithm of the probability is represented as an integer of which the lower bits denote the bits after the decimal point. Thus we have to design a binary arithmetic coder that can handle a probability distribution represented as logarithms. It has to fulfil the following requirements:

- **Performance:** The optimal length of the code word for a source sequence with probability $P_c(x_1^N)$ is $-\log_2 P_C(x_1^N)$ bits. In theory an arithmetic coder can be constructed that has a redundancy of less than two bits on the entire source sequence compared to the optimal length. This arithmetic coder requires computations of infinite accuracy, and cannot be used in practice. A practical arithmetic coder will have to round the probabilities, which will increase the redundancy. But a good arithmetic coder should be able to perform close to optimal. We require that our implementation of the arithmetic coder introduces a redundancy of less than 0.01 bit per source symbol of 8 bits.

- **Complexity:** An arithmetic coder uses a few registers, e.g., to store the probabilities and the lower boundary point of the interval. We require that our implementation uses a negligible amount of memory.

- **Speed:** The data compression program has to achieve a speed of 10 kBytes/second, or 80 kbits/second. Since the CTW algorithm is much more complex than the arithmetic coder, we will allocate only a small portion of the available time to the arithmetic coder. We require that the arithmetic coder may use at most 10 % of the available time, therefore it should achieve a compression speed of at least 800 kbits/second.

Chapter 3 discusses the design of an arithmetic coder that fulfils these requirements. The final design is an implementation of the Witten-Neal-Cleary coder with 16 bits accuracy that accepts a binary probability distribution in a fixed-point logarithmic representation as input. This arithmetic coder has been assessed on three test sequences. Its redundancy is around 0.0001 bits/symbol, its compression speed is greater than 850 kbits/second and it uses only a few registers and a small table (just 3072 records of two bytes), so the memory requirements are negligible. Therefore this subsystem meets its goal and achieves all three requirements.

THE BASIC DESIGN: THE CTW ALGORITHM

The CTW algorithm should estimate the probability distribution of each source symbol. This estimate is then used in the arithmetic coder. Since the performance of the arithmetic coder is almost optimal, the performance of the entire data compression algorithm depends solely on the CTW algorithm. The algorithm has to fulfil the following requirements.

- **Performance:** In combination with the arithmetic coder it should at least achieve the weak performance criterion, but preferably the strong performance criterion. Thus the performance of CTW should be at least as good as "state selection" and PPMZ-2 on the entire Calgary corpus. Preferably, it should also compress each of the four text files

within 0.02 bits/symbol of these two competitive algorithms, and each other file within 0.04 bits/symbol.

- **Complexity:** The CTW algorithm should not use more than 32 MBytes of memory.

- **Speed:** The CTW algorithm, together with the arithmetic coder, should achieve a compression speed of at least 10 kBytes/second on our reference system.

Chapter 4 discusses the design of the CTW algorithm. Our design uses a forward decomposition to decompose the source symbols. The context trees will be stored in a hash table of 32 MBytes. The records require only 6 bytes each, thus the hash table can contain almost 5.6 million records. Once the hash table is filled, the context trees will simply be frozen. The number of records that needs to be created is significantly reduced by a unique path pruning technique. In our implementation this technique requires a small additional buffer to store the most recent part of the source sequence. The computations are performed in a fixed-point logarithmic implementation and many computations are replaced by simple table-lookups. Finally, the performance has been improved by fine-tuning several variables, e.g. the estimator.

This implementation of the CTW algorithm uses slightly more than 32 MBytes of memory, thus it meets the memory criterion (although for `book1` the hash table is too small). Its performance on the four text files is better than "state selection" and comparable to PPMZ-2. But its performance on the other files of the Calgary corpus is worse by more than 0.04 bits/symbol, compared to PPMZ-2. Furthermore, it achieves a compression speed between 6 and 8 kBytes/sec. (6 kBytes/sec. on average on the Calgary corpus, 7 kBytes/sec. without `book1`). Thus it does not meet the performance and speed criteria yet.

2.3.2 STEP 2: THE FORWARD DECOMPOSITION

The basic CTW algorithm estimates the probabilities of individual bits. It uses the straightforward ASCII decomposition (see Section 4.3.1) to translate source symbols into 8 bits, each bit has its own context tree, and consequently each symbol requires 8 estimations and thus 8 updates of the context trees. The number of updates is proportional to the memory usage and the computational complexity of the CTW algorithm. The average number of updates per symbol can be minimized by computing for each source sequence a special forward decomposition that minimizes the average number of bits per symbol. This will help the final design to meet the speed criterion. But it has two disadvantages. First, it requires an additional pass over the source sequence. Although sequential coding is not a crucial feature of our CTW implementation, we are reluctant to give it up, unless it results in a significant computational complexity reduction. Secondly, the encoder has to describe the forward decomposition for the decoder. More precisely, a special forward decomposition has to fulfil the following requirements.

- **Performance:** The CTW algorithm with the special forward decomposition has to result in at least the same performance as the CTW algorithm with an ASCII decomposition, even though the special forward decomposition has to be described too.

- **Speed:** The forward decomposition should result in a significant increase in compression speed. At depth 10, and without `book1` (for which the hash table is too small), the current design achieves about 7 kBytes/sec., so the average speed should increase by about 40 % to meet the speed requirement.

- **Complexity:** The hash table is limited to 32 MBytes, therefore the memory criterion will be met. But at depth 10, `book1` requires about 8.3 million records, while the hash table can only hold 5.6 million records. To fully capture the entire model for `book1`, the number of records should decrease by about 30 %.

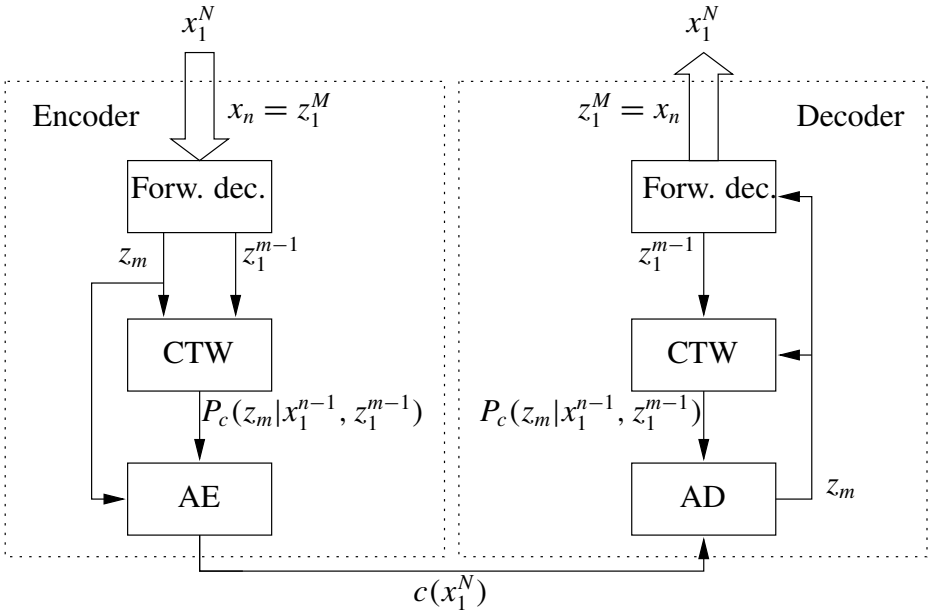


Figure 2.2: The CTW design with a forward decomposition.

The new design is shown in Figure 2.2. The encoder first computes the special forward decomposition. Then it translates each source symbol x_n into a variable number of bits depending on the decomposition. Suppose symbol x_n is described with M bits, $z_1, z_2, \ldots, z_M$[1]. The CTW algorithm receives the value of the next bit z_m, and the values of the previous bits z_1^{m-1} of this symbol, such that it knows which context tree it has to use to compress bit z_m. The decoder first reads the special forward decomposition and then uses this decomposition to translate the individual bits back into the original source symbols.

Chapter 5 discusses the design of a special forward decomposition. Our final design uses the decomposition called "Huf-IV". The depths of the symbols in this decomposition are computed

[1]Because the number of bits, and the values of the bits, depend on the value of x_n, we should actually use a more elaborate notation: symbol x_n is decomposed in some M_n bits, $z_{n,1}, \ldots, z_{n,M_n}$. But for sake of simplicity, we use the shorter notation in this thesis.

from the Huffman code for the source sequence, and the place of the symbols in the decomposition is obtained by a sorting technique that uses the initial values of the symbols. This special forward decomposition reduces the number of records by about 30 % (and by more than 40 % on book1), it increases the compression speed by at least 25 % on most files, and almost doubles the average compression speed on the entire corpus to 11.9 kBytes/sec., while it also improves the compression performance by about 1 %.

The CTW program with the Huf-IV forward decomposition uses slightly more than 32 MBytes of memory. It compresses the longer files of the Calgary corpus at about 10 kBytes/sec., and achieves an average speed of more than 10 kBytes/sec. It has a better performance on the four text files than PPMZ-2, and a slightly better average performance on the reduced Calgary corpus (the entire corpus except files geo and pic). Therefore, this design meets the weak performance criterion plus all other key design criteria.

2.3.3 STEP 3: SWITCHING

There is a new trend towards data compression programs that combine different compression algorithms. Because these algorithms have different characteristics, they will perform best on different types of files. By cleverly combining these algorithms, such programs are in general better equipped to achieve a good compression rate on a wide range of files. We investigated whether one of these new combining techniques can be used in our final design to improve its performance such that it will fulfil the strong performance criterion too, at the cost of only a small increase in memory and computational complexity.

There are two basic techniques to combine different data compression algorithms in one program. First, the combining program can analyse the source sequence, and then decide which algorithm it will use to compress the entire sequence. Such a program can at best achieve global optimality: the compression rate on a source sequence can be as good as the best compression rate achieved by all algorithms it combines. Alternatively, the combining program could run different algorithms simultaneously, and use on each *part* of the source sequence the algorithm that performs locally best. Now it is possible that the combining program compresses a source sequence to a compression rate that is lower than that of each of the algorithms it combines. Because, the latter technique is more powerful, we decided to investigate one of its newest implementations, the switching method, here. It combines two data compression algorithms, therefore, such system requires two additional subsystems in our design: the snake algorithm and the companion algorithm. The snake algorithm is an efficient implementation of the switching method, and the companion algorithm is the second data compression algorithm that will run in parallel with CTW. Now, our design consists of five subsystems (Figure 2.3 shows the encoder; the decoder has to be extended similarly).

SWITCHING: THE SNAKE ALGORITHM

Suppose the forward decomposition splits source symbol x_n in M bits, $z_1, \ldots, z_M$. The snake algorithm receives for every bit z_m, with $m = 1, \ldots, M$, two estimates of the probability distribution, $P_{\mathcal{A}}(z_m|x_1^{n-1}, z_1^{m-1})$ from CTW and $P_{\mathcal{B}}(z_m|x_1^{n-1}, z_1^{m-1})$ from the companion algorithm. From

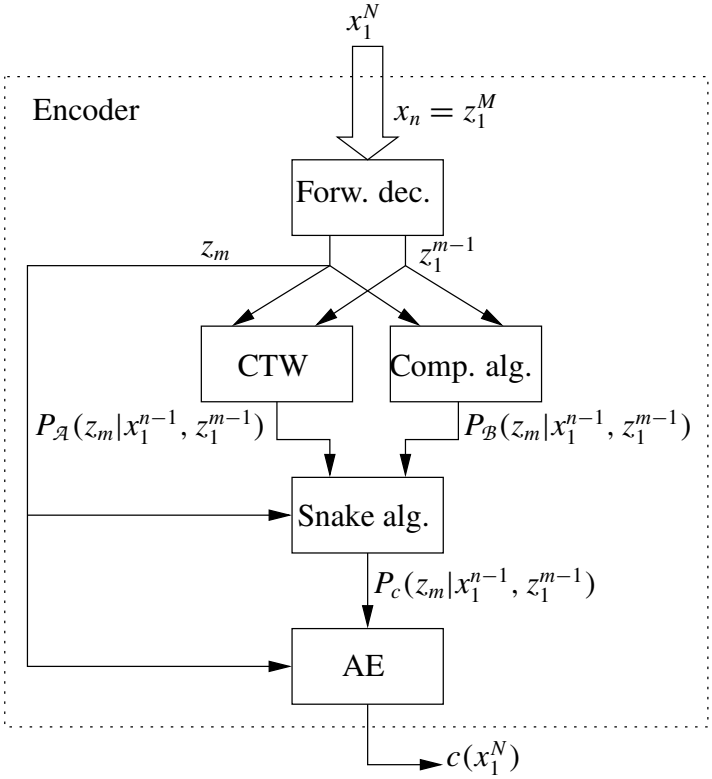


Figure 2.3: The final design

these two estimates, it computes the coding probability distribution $P_c(z_m|x_1^{n-1}, z_1^{m-1})$ of this bit for the arithmetic coder.

The CTW algorithm already uses 32 MBytes of memory, and it achieves a compression speed of about 10 kBytes/second. In order to also achieve the strong performance criterion, we are willing to loosen the requirements on the memory complexity and the compression speed slightly. But it is a trade-off, in which the gain in performance has to be weighed against the increase in complexity. Therefore, it is very difficult to set strict constraints on the amount of time and memory that these additional subsystems may use and consequently, it is even more difficult to set such constraints for the snake algorithm alone.

- **Performance:** The snake algorithm together with the companion algorithm, should improve the performance of CTW, such that the strong performance criterion is achieved. Therefore, for all files of the Calgary corpus it should perform within 0.04 bits/symbol of the best competitive algorithms, "state selection" and PPMZ-2.

- **Complexity/Speed:** The additional memory usage and the decrease in compression speed, due to the snake algorithm together with the companion algorithm, should be so small that the improvement in performance justifies their usage. This trade-off can only be evaluated after completion of this step of the design process.

Chapter 6 discusses the design of the snake algorithm. The snake algorithm weights both estimates for the probability distribution of the next symbol, such that the algorithm that performs locally best will dominate. The final design of the snake algorithm uses a fixed-point log representation for the floating point numbers. It uses almost no memory and it reduces the compression speed of the CTW algorithm by only 10 %. Thus the additional complexity caused by the snake algorithm alone seems reasonable. But because the snake algorithm is a greedy approximation of the switching algorithm we cannot guarantee that it is able to switch effectively between two data compression algorithms. This needs to be verified once the companion algorithm has been designed and implemented.

SWITCHING: THE COMPANION ALGORITHM

The snake algorithm can achieve its highest gain if the two data compression algorithms between which it can switch have a complementary local performance. Because the CTW algorithm is fixed now, we should design a companion algorithm that is complementary to CTW. One of the weaker points of CTW is its performance on long repetitions and we decided to focus on this point. There are several algorithms, e.g. Lempel-Ziv type algorithms and PPM algorithms with a high maximum depth, that have a good performance on the long repetitive strings. Because it is easier to integrate PPM algorithms into our final design than Lempel-Ziv algorithms, we chose to use a PPM algorithm as our companion algorithm.

The companion algorithm, together with the snake algorithm, has to fulfil the following requirements.

- **Performance:** Together with the snake algorithm, it should improve the performance of the final design. The complete system should fulfil the strong performance criterion.
- **Complexity/Speed:** The additional memory usage and the decrease in compression speed, due to the companion algorithm together with the snake algorithm, should be so small that the improvement in performance justifies their usage.

Chapter 7 discusses the design of the companion algorithm. Because the companion algorithm will only be used in parallel to CTW, its performance on a source sequence is not important, but only its ability to encode long repetitions efficiently matters. Consequently, we were able to both bias the companion algorithm even more in favour of long repetitions by increasing its maximum context length, and to greatly reduce the complexity of the algorithm. The final design implements PPMA. In order to reduce complexity, we modified it such that it uses only the already available information from the context trees for contexts of lengths up to and including D. Furthermore, to improve the performance, it has been extended to use much longer contexts and this extension requires an additional buffer of 2 MBytes. The combination of the snake algorithm and the companion algorithm improve the performance of CTW by more than 1 %, and it now meets the strong performance criterion. But, the complete system requires about 35 MBytes of memory, 10 % more than allowed, and more importantly, the compression speed is now reduced from 11.9 kBytes/sec. to only 6.6 kBytes/sec., far below the compression speed criterion.

Still, the snake algorithm and the companion algorithm are a valuable addition to the data compression program. The computer technology has improved enormously since the start of the project. PPMZ-2 uses this additional computational power. It is slower than our entire final design, and it uses up to 100 MBytes of memory. Therefore, it seems best to implement the snake algorithm and PPM as a user option. Now the user can choose between a system, CTW alone, that achieves a very good performance, and meets all speed and memory requirements. Or the user can use the system with PPMA and the snake algorithm, which has a performance which is at least comparable to the best state-of-the-art programs, but which uses slightly more than 32 MBytes of memory, and which is about 34 % slower than the compression speed criterion. In the end, by offering users a choice between these two set-ups, we may conclude that we have completed the project successfully.

2.4 OTHER INVESTIGATED FEATURES AND CRITERIA

Three features or possible criteria of the final design have been investigated during the design process, but in the end it was decided not to implement these in the final design. In this section we will briefly describe some of the work and ideas concerning these features.

2.4.1 PARALLEL PROCESSING

There are two possible directions for allowing parallel processing in the CTW algorithm. We could either try to fit as much parallelism as possible into the CTW algorithm as it is at the moment. Or we could start with the assumption that we have multiple processors, and develop a CTW-like algorithm for this situation. This section describes two ideas for the first direction, and three for the second direction.

In order to understand the limitations caused by the CTW algorithm, it is important to briefly describe the order of the computations in the estimation process. CTW collects its statistics in a context tree, which has a fixed maximum depth, say D. The context, the previous D symbols in the source sequence, specify one leaf of this tree. Coding, or more exactly the estimation, is done by estimating at each depth the probability of the new symbol, and then by weighting these estimates together. CTW starts at the highest depth, estimates the probability, updates the node, and continues with the parent node of this node. This process repeats itself until it reaches the root node which gives the final estimated probability distribution for the new source symbol. A node in a context tree can only be used for coding the next symbol if is has been updated for all previous symbols. This sets a fixed order in which a node has to be updated throughout the source sequence. As a result, this fixed order limits the maximum number of processors that can be used simultaneously.

Without modifications, multiple processors can be effectively used in the encoder of the CTW algorithm.

I The first technique introduces some sort of pipelining. As soon as a node has been updated for a certain source symbol it can (if necessary) be used to encode the next symbol. Thus

while one processor is estimating the probabilities at depth 0 of the context tree for the current source symbol, a second processor could already estimate the probability at depth 1 of the context tree for the next symbol. Therefore, for a context tree of depth D, in total $D+1$ processors could work simultaneously.

II The second technique only applies to non-binary source symbols. As will be explained in Section 4.3.1 the CTW algorithm uses different context trees for coding each bit of a source symbol. Since the trees are updated independently, the updating, and thus the coding can be done concurrently.

Both techniques are shown in Figure 2.4 for symbols consisting of 2 bits and context trees with maximum depth $D = 2$. Processors μ_1, μ_3, and μ_5 can work simultaneously because they operate on nodes in the same context tree, but at different depths (technique I), and at the same time processors μ_2, μ_4, and μ_6 can process the second bit (technique II). In total 6 processors ($(D+1)\cdot M$, in which M is the number of bits per source symbol) can work in parallel without any synchronization problems.

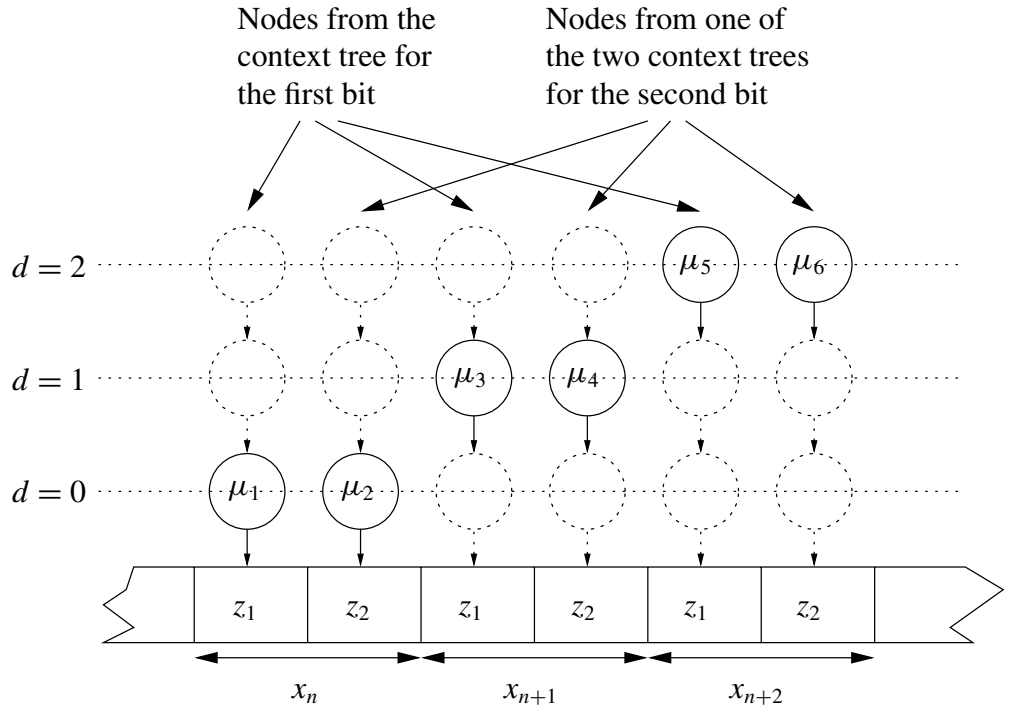


Figure 2.4: Parallel encoding with the CTW algorithm.

Techniques I and II cannot be used for the decoder. The first technique cannot be applied, because only after decoding the current symbol, the CTW algorithm can update the statistics in the context tree. But also the second technique cannot be applied because the CTW algorithm will use different context trees for the second bit of a symbol, depending on the value of the first bit. Since the decoder cannot look ahead in the source sequence like the encoder, it is very

difficult to implement concurrent computing without modifying the CTW algorithm. Therefore, we investigated some ideas to redesign the CTW algorithm in order to simplify the parallel encoding and decoding.

III The most straightforward idea is to split the source sequence in blocks and let each processor compress its own block, independently of the other blocks. This is shown on the left hand side of Figure 2.5 for two processors. Clearly, such a parallel CTW algorithm can be implemented easily. But this modification will have a huge impact on the performance. This is depicted in the stylized figure below the block diagram. It shows for each position in the source sequence the local compression rate achieved by the parallel CTW algorithm (the solid line) and by the regular CTW algorithm (the dotted line). The CTW algorithm is adaptive, so at the start of the source sequence the compression rate is poor, but it will rapidly converge towards its final compression rate. Since both blocks are compressed independently now, the parallel CTW algorithm has to adapt to the source sequence in both blocks: the "learning costs" are being paid twice. The difference between the solid line and the dotted line is the loss in performance. From a technical point of view, each block has its own set of context trees, therefore the statistics of the source sequence will be spread over more trees than before, and as a result the estimates will be less accurate.

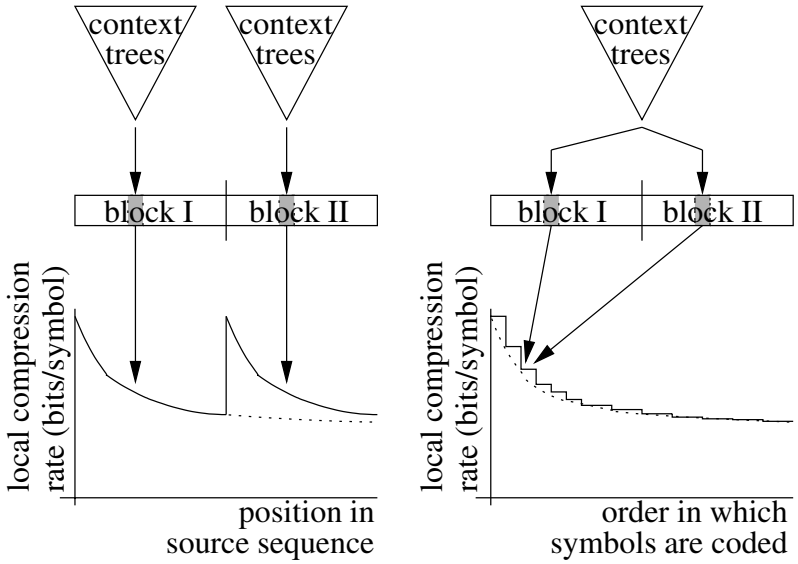


Figure 2.5: Parallel coding with the CTW algorithm.

IV An improved parallel CTW algorithm is depicted on the right hand side of Figure 2.5. It uses only one set of context trees. The context trees are "frozen", as both processors code the next symbol in their block. Then, in one step, the context trees are updated for both source symbols. The advantage is clear, after every two source symbols the context trees are fully up-to-date. This is also visible in the stylized performance plot. In this plot the symbols are now ordered on the moment at which they are coded. In order to look at this parallel

CTW algorithm from a different perspective, consider the sequential CTW algorithm that alternately encodes a symbol from both blocks. Such a sequential algorithm would achieve exactly the same performance as the regular sequential CTW algorithm, minus some small initialization losses for the second block. If this new sequential CTW algorithm is compared to the parallel CTW algorithm, then they behave identical, except that for the symbols in block II, the parallel CTW algorithm misses *one* update (the most recent symbol from block I). This results in a very small additional overall loss, which is indicated by the small area between the dotted line and the solid line in the performance plot in Figure 2.5. But this parallel CTW algorithm has an other disadvantage: the two processors will share at least one, but possibly more nodes, in the context trees. This might cause problems in the updating phase, because these nodes have to be updated for both symbols. This problem gets worse if more processors are being used. Then several nodes will have to be updated multiple times. Fortunately, most of these nodes will be in the lower parts of the context trees, and the coding of the new symbols will start in the upper part of the context trees. Therefore, one could collect all the multiple updates in a queue and use some dedicated processors to work through this backlog in updates, while simultaneously the other processors already start coding the new symbols. Such an algorithm would require additional administration in the nodes and additional communication between the different processors, so the gain in compression speed might not be proportional to the number of involved processors.

V An elegant solution can be found by running the algorithms in the different blocks in a different phase. E.g., if the source symbols consist of two bits, then we can use two blocks. Assume that we decide to let the coding in block II run exactly one bit behind the coding in block I. Suppose, the CTW algorithm in block I should now code the second bit of, say, the eighth symbol. Then, the context tree for the first bit is fully updated for the eighth symbol in block I, and the CTW algorithm in block II can use it to code the first bit of its eighth symbol. In the next step, the CTW algorithm in block I can start with the first bit of its ninth symbol, with a context tree that now also includes the update of the first bit of the eighth symbol in block II. And, in block II, the CTW algorithm can compress the second bit of its eighth symbol, again, with a fully updated context tree. This technique has two advantages over technique IV. First, because the CTW algorithms in the different blocks process a different bit, they work in a different context tree, and their updates will never collide. Secondly, the context trees are always fully updated, thus the performance will be as good as the sequential CTW algorithm (except for some small initialization costs for the extra blocks). The drawback of this idea is that the source sequence can be split in at most the same number of blocks as the number of bits in each source symbol.

Especially the fifth technique looks promising because it improves the speed of both the encoder and decoder, at a very low cost in performance. Still, we decided not to implement such a parallel CTW algorithm, because it is not an essential feature of our design, and the design time necessary to complete a parallel implementation had to be spent on achieving the key design criteria.

2.4.2 RANDOM ACCESS, SEARCHING AND SEGMENTATION

RANDOM ACCESS

Random access in the regular CTW algorithm is probably impossible. Therefore, we investigated different modifications in order to allow random access. Our investigations led us in two directions: either working from the bottom up, or looking from the top down.

We started at the bottom by trying to develop a good random-access code for memoryless sequences. The performance of this code should be as close as possible to the performance of the Krichevsky-Trofimov estimator, which is used in the CTW algorithm to code memoryless sequences. This resulted in a random coding scheme [17, 67], which splits the source sequence into blocks, such that the decoder can decode one of these blocks without decoding the other ones. The scheme is based on enumerative coding. In enumerative encoding [14, 36] one first codes the parameter, followed by a description of the sequence given this parameter. Similarly, in our scheme the source sequence is coded in a series of parameter blocks and descriptive blocks. Each descriptive block corresponds to one block in the source sequence. The parameter blocks form a hierarchical structure, such that from a logarithmic number of parameter blocks the parameter for one descriptive block can be computed. One of the main challenges while developing this scheme was to be able to compute the starting points of the descriptive block and the parameter blocks in the compressed sequence without having to maintain a complete index. This scheme has a redundancy of only a fixed number of bits per block. Unfortunately, it was very difficult to implement this scheme in practice due to the very high accuracy required for the computations [17]. Therefore, this scheme is not a feasible "random access" substitute for the regular memoryless estimators yet.

Alternatively, we started at the top by investigating the context-tree maximizing algorithm [50, 52, 53] (see also Appendix B). The maximizing algorithm finds the model that minimizes the description length, the MDL model. The description length of a source sequence is the length of the description of a model plus the length of the description of the source sequence, given that model. A compression program based on the maximizing algorithm is always a two-pass program. In the first pass it computes the MDL model for the source sequence and in the second pass it encodes the MDL model first, followed by the encoding of the source sequence itself. This algorithm could be more suitable for random-access than CTW, because it freezes the model and it separates the model description from the description of the source sequence (analogous to enumerative encoding). Suppose, we again want to split the source sequence into blocks, then we can encode the model description in various ways. We can e.g., describe the entire MDL model in advance, describe a separate MDL model for each block, or describe one core model (maybe even including some parameter information) that is used for every block, plus for each block a description of a refinement on this core model. Alternatively, we can order the blocks in some hierarchy, such that each block can be decoded after decoding at most a logarithmic number of other blocks. These schemes focus mostly on describing the model efficiently, but also the source sequence has to be described. The MDL model effectively splits the source sequence in memoryless subsequences, but the memoryless subsequences can have a variable number of symbols in each block of the source sequence. Therefore, this problem is similar to, but also more

complex than the problem described in the previous paragraph, which we couldn't satisfactory solve. Ultimately, one could decide to accept the losses and let the parameters adapt in each block individually. Unfortunately, experiments show that the performance of the maximizing algorithm is already considerably worse than the performance of the CTW algorithm and some of the improvements of the maximizing algorithm [52, 53] (e.g. on-the-fly, which mixes the model description and the description of the source sequence again) might not be possible in combination with random access. If random access causes a further degradation in performance, it might not be possible to achieve the performance criterion.

In the end we decided not to implement these schemes. There is no comprehensive theory for random access. There are no known bounds on the minimum additional redundancy induced by random access and currently, everything beyond the most straightforward schemes still needs a lot of investigation before it can be applied in a practical design.

SEARCHING

Searching has not been investigated. If the compressed data has to be searched, without the availability of an index, then this problem is related to random access and, as we described above, we were not able to include random access in the final design. A compression program that allows random access and searching is described in [70]. It has been achieved by using a fixed code instead of an adaptive algorithm.

SEGMENTATION

There are several straightforward ways to implement segmentation. For example, an algorithm can analyse the data in detail, and choose for every segment the best data compression algorithm from a given set of algorithms, or the best set-up of one specific algorithm, like CTW. For the CTW algorithm, segmentation is important because CTW is a stationary algorithm. If the source is dynamic, and the parameters change significantly throughout the source sequence the performance of CTW might be poor. Therefore it would be useful to split the sequence in segments in which the source behaves more or less stationary. But it is very difficult to identify in the source sequence where one segment ends and a new one starts.

The switching method (see Chapter 6) is a partial implementation of a segmentation strategy. It uses two data compression algorithms, the CTW algorithm and a Lempel-Ziv-like algorithm, that are running in parallel, and it allows a switch from one algorithm to the other one between any two source symbols. It is not an exact implementation of the segmentation behaviour, because both adaptive data compression algorithms observe the entire source sequence, thus also the segments on which the other algorithm is used. In a real segmentation implementation you might want to stop the adaptive algorithm that is currently not used. The final design uses an implementation of the switching method, thus segmentation has been achieved in the final design to some extent.

Different approaches to segmentation, and running multiple data compression algorithms in parallel have been discussed in [69].

2.4.3 FLEXIBILITY

Texts have been explicitly mentioned in the project description. Therefore, we investigated a possible extension of the CTW algorithms for texts. The CTW algorithm uses only the last, say 10, symbols of the source sequence to estimate the probability distribution of the next source symbol. But of course a text has a lot more structure, like e.g. grammar. Grammatical information can be added explicitly to a text by attaching a tag to each word. A tag could be a rather global specifier, like "noun", "verb", etc., or it could be more detailed, like "plural masculine noun". These tags can be either added to the text by hand, or automatically with taggers. This last option leads to an interesting semi-two-pass strategy: in the "first pass" a tagger is used to label all the words, and possibly the punctuation, in the text, while in the second pass the text plus tags is being compressed. It has been reported that compressing a text with tags results in a shorter code word length than compressing the text alone [43, 44]. More in depth investigations on a CTW algorithm with tagged texts [6], shows more subtle differences. First of all, it is difficult to make a fair comparison, because compressing the tags induces additional complexity. Thus a Tag-CTW algorithm should be compared with a regular CTW algorithm with the same complexity, and consequently with a higher maximum context length. Also, because the Tag-CTW has more information, it will take longer before it converges to its best compression rate, therefore the algorithm should be compared on long texts (a few megabytes). From our initial investigations we concluded, that adding tags sometimes increases the performance slightly, say about 1 % on long texts, while in general it will not degrade the performance.

Of course, the Tag-CTW could be further improved and optimized in order to gain more than 1 %, but still we decided not to use tags in our final design, because the gain is relatively small, and it would require that a complete tagger is added to the final design.

2.4.4 A HARDWARE IMPLEMENTATION

A hardware implementation of the CTW algorithm has not been mentioned in the project description. A hardware implementation is aimed at a different type of applications. While a software compression utility is cheap and can be used on existing computer systems, it will also be rather slow in general. Thus if an application requires a really high compression and decompression speed, then a hardware implementation could be useful. Hardware compression can be used inside communication equipment for computer networks (high speed modems, Ethernet cards, etc.), and inside storage equipment (tape streamers, hard disks), but also as an additional feature built into computer equipment, that can be used by any software package requiring the processing of huge quantities of data, like databases, or audio and video editing software. Before such a hardware implementation can be developed all properties of the device have to be investigated in detail. What format will the data be stored in? Should it be compatible to existing software or hardware solutions? Should one be able to randomly access the data? Should the data be protected against error-propagation? How fast should it be? After evaluating these criteria it might, for some purposes, be cheaper to use existing general purpose processors than to develop an application specific IC. All these considerations are beyond the scope of this thesis, but still we studied some possibilities for the CTW algorithm.

Most existing hardware implementations of data compression algorithms are based on the Lempel-Ziv algorithms, e.g. the V42bis modem standard. The Lempel-Ziv algorithm processes the source sequence sequentially. The most straightforward hardware implementation of such an algorithm can use dedicated hardware to perform each step in the process sequentially, but very fast. The most time consuming step in a regular Lempel-Ziv implementation is the string matching step in the compressor. In this step a string, the next few symbols of the source sequence, is compared to a library consisting of a set of previously observed strings. This step can be implemented in parallel in hardware, which could result in a very significant gain in compression speed. There are various ways to implement this step in parallel in hardware, e.g. with a systolic array [40], or with associative arrays [8].

At the moment there has not been so much research on hardware implementations of adaptive algorithms, like CTW. The same considerations as for the dictionary based Lempel-Ziv algorithms apply. One could either design dedicated hardware to speed up the sequential operations, or one could try to redesign the algorithm in order to allow parallel processing in the hardware implementation. For this last option some of the parallel implementations of CTW as discussed before seem quite promising. Both the solution with processors working through the back-log in updates while the coding for the new source symbols has already started (technique IV), and especially the solution with the out-of-phase blocks (technique V), seem feasible in a hardware implementation. One step in the coding process of CTW has not been mentioned in the discussion of the parallel CTW implementation. Before a source symbol can be coded, first the correct path in the context tree has to be found. While of course an efficient searchable data structure can be applied, there are other options in a hardware implementation. For example, one can use associative memories. These memories consist of a decoder and the actual memory. Through the decoder these memories can not only be accessed based on their addresses, but also based on the contents of the memory cells. A key is provided to the memory decoder, and all memory cells that, according to some predefined relation, match that key, will offer their content serially on a data bus. One can imagine that, by offering a single key, in one operation all records on the context path are activated. In this way the speed can be increased even further.

We didn't study the design of a hardware implementation in detail, because a hardware design is not a part of our project description, but it is one of the angles from which we wanted to look at our design.

2.5 THE DESIGN PROCESS IN RETROSPECTIVE

When the project started, two previous designs and their ideas were available. One design already achieved the weak performance criterion at that moment, but it used too much memory and compression time. The other design was faster and used less memory, it already met the memory criterion, but its performance was slightly worse. It was expected that a combination of the ideas used in these two designs would quickly result in a solid basis for our final design. Therefore, originally we intended to investigate the design of the data compression program from a broad perspective: e.g., random access, parallel processing, and maybe even some initial work on a hardware implementation.

During the design process, we had to shift our focus more towards achieving the three key criteria. On one hand the performance of the competitive algorithms improved faster than expected, while on the other hand, it was more difficult to achieve the speed criterion than expected. This also explains our three step design process. After the first step, the design did not fulfil the performance criterion and the speed criterion. The objective of the second step was mainly to improve the compression speed, while the third step was aimed at improving the performance. In our design process the design was completed after each step and then evaluated. Only after the evaluation we could decide whether it was necessary to extend the design or not. The advantage of such a design process is that each step uses the completed design from the previous step as its basis and therefore the integration of the components will be simple. The disadvantage is that it is difficult to prepare for subsequent design steps, because only after evaluation of this design it will be clear which extension, if any, is necessary. As a result, design decisions are mostly made to improve the current design, but they could hamper following design steps. A prime example is the fixed data structure of the context trees, optimized for the CTW algorithm during the first design step, that later limits the choice of the escape mechanism for the companion algorithm in the third step (as discussed in Chapter 7).

Furthermore, after completion of the project, it appears that some of the design decisions made early in the project might have to be reconsidered, with the knowledge that is available now. For example, in Chapter 4 we rejected the multi-alphabet CTW algorithm, because its computational complexity was to high. Later investigations [60] showed that some of the most complex steps can probably be greatly simplified. A multi-alphabet CTW could have greatly simplified the design of the snake algorithm and companion algorithm. But, such things are inevitable for long design processes in a rapidly changing field.

Finally, we underestimated the great difficulties that still need to be overcome before additional features, like random access and parallel processing can be efficiently implemented. Most work on these features had to start from the very beginning, and although some progress has been made, it will still take a considerable amount of research before such ideas can be implemented in a practical design.

During this project we have developed many innovative ideas. Some of these ideas have been used in the final design (the special forward decomposition, switching, and the decomposition of PPM, among others), but others were not well suited for our final design. Appendix B briefly describes some of the results obtained during this project that have not been used in our final design. Most of the new ideas have been published in papers. Ultimately, we realized an innovative design, split up in two programs, that meets its goal and achieves all key criteria. We believe that the project has been concluded successfully.

3

SUBSYSTEM 1: ARITHMETIC CODER

This chapter discusses the implementation of the arithmetic coder for the final design. Based on the theory described in Chapter 1, two coders are investigated and evaluated on speed and performance. Both coders meet the design criteria, and in the end the Witten-Neal-Cleary coder is chosen because it has a marginally smaller redundancy.

3.1 DESCRIPTION

THIS chapter is concerned with the design of an arithmetic coder. The arithmetic coder gets a binary probability distribution in a fixed-point log-representation from the CTW algorithm. The final implementation has to fulfil the following requirements:

- The final design should achieve a code word length that is at most 0.01 bits/symbol longer than the optimal code word length of $-\log_2 P(x_1^N)$ bits, in which $P(x_1^N)$ is the coding probability of the source sequence[1].
- The final design should use only a negligible amount of memory.
- The final design should achieve a compression speed of at least 800 kbits/second.

In this chapter two practical binary arithmetic coders will be described, the Rubin coder and the Witten-Neal-Cleary coder, and the resulting implementations will be tested on three test sequences to check whether they fulfil these criteria.

[1] In this chapter we omitted the subscript and $P(x_1^N)$ denotes $P_c(x_1^N)$

3.2 BASIC ALGORITHM

The basic arithmetic coding algorithm, called the Elias algorithm (see [21]), has already been described in Section 1.5.5 for binary source symbols. The encoder is a direct implementation of the update rules and the length assignment. The following steps describe the binary encoder.

- Initialization: $P(\emptyset) = 1$, and $Q(\emptyset) = 0$.
- For all x_n, with $n = 1, 2, \ldots, N$, do:
  $P(x_1^n) = P(x_1^{n-1}) \cdot P(X_n = x_n | x_1^{n-1})$
  $Q(x_1^n) = Q(x_1^{n-1}) + P(x_1^{n-1}) \cdot P(X_n < x_n | x_1^{n-1})$
- $c(x_1^N) = \lceil Q(x_1^N) \cdot 2^{l(x^N)} \rceil \cdot 2^{-l(x^N)}$, with $l(x^N) = \lceil -\log_2 P(x_1^N) \rceil + 1$.

The Elias algorithm results in a code word length $l(x_1^N)$ which is at most two bits above the optimal length. The encoder uses only two floating point numbers ($P(.)$ and $Q(.)$), and at most one addition and two multiplications per source symbol.

The decoder first determines a threshold value $T(.)$. The threshold is the exact point at which the interval will be split in the next step. If the received code word $c(x_1^N)$ is below this threshold, then the next symbol is a '0', otherwise it is a '1'.

- Initialization: $P(\emptyset) = 1$, $Q(\emptyset) = 0$.
- For all x_n, with $n = 1, 2, \ldots, N$, do:
  - $T(x_1^{n-1}) = Q(x_1^{n-1}, x_n = 1) = Q(x_1^{n-1}) + P(x_1^{n-1}) \cdot P(X_n = 0 | x_1^{n-1})$
  - If $c(x_1^N) < T(x_1^{n-1})$ then $x_n = 0$, else $x_n = 1$.
  - $P(x_1^n) = P(x_1^{n-1}) \cdot P(X_n = x_n | x_1^{n-1})$
    $Q(x_1^n) = Q(x_1^{n-1}) + P(x_1^{n-1}) \cdot P(X_n < x_n | x_1^{n-1})$

3.3 ALGORITHMIC SOLUTIONS FOR AN INITIAL PROBLEM

3.3.1 FUNDAMENTAL PROBLEM

The Elias algorithm has a fundamental problem, which prevents it from being implemented as described in the previous section. It assumes that the computations are carried out with infinite accuracy and as a result, registers $P(.)$, $Q(.)$, and $T(.)$ can become infinitely long. This problem has been solved in the latter part of the 1970s. In 1976 Pasco [30] described a rounding technique to fix the size of the register for $P(.)$, but it still required an unbounded register for the lower boundary point. Also in 1976 Rissanen [31] published an arithmetic coder with fixed length registers, but this coder creates the code word from right to left, thus the encoding can only start after the entire source sequence is observed. In 1979 Rissanen and Langdon [33] mentioned that they had an arithmetic coder that uses only fixed length registers and that constructs the code

word from left to right. They applied for a patent on this technique, which was granted in 1984 [23]. But, also in 1979 Rubin [35] published his practical implementation of an arithmetic coder with fixed length registers only.

First the length of the register $P(.)$ containing the interval size will be fixed. The size of the interval, the probability of the sequence so far, monotonically decreases. Thus the representation of the interval size will start with a growing number of zeros. Furthermore, the number of significant bits following these zeros can be fixed to, say, F bits by rounding after each operation at the cost of a small loss in performance. Now, a register of fixed-size F can be used to represent the size of the interval as follows. Before the new symbol is encoded, this register will be scaled until it uses its full accuracy by shifting out the leading zeros. In other words, this register containing the significant bits will be shifted to the right along the (infinitely long) register containing the lower boundary point. Next, the interval can be split into two subintervals, each subinterval size rounded to an accuracy of F bits. The new interval size will then still fit in the fixed size register.

Even with a fixed-size register for the interval size, the representation of the lower boundary point still grows with the length of the sequence. Suppose the probability is indeed truncated to F significant bits. The new symbol will split this probability into two parts, and in case the new symbol is a '1', the probability in case the symbol was a '0' has to be added to the lower boundary point. Suppose we already scaled k times, and let us denote the representation of the lower boundary point by $0.q_1q_2\ldots q_{k+F}$ and the representation of the significant bits of the probability by $p_1p_2\ldots p_F$. Then the new lower boundary point is:

$$\begin{array}{llllllllll} 0. & q_1 & q_2 & \ldots & q_k & q_{k+1} & q_{k+2} & \ldots & q_{k+F} & \\ 0. & 0 & 0 & \ldots & 0 & p_1 & p_2 & \ldots & p_F & + \\ \hline 0. & q'_1 & q'_2 & \ldots & q'_k & q'_{k+1} & q'_{k+2} & \ldots & q'_{k+F}. & \end{array}$$

Now, it is clear that the first k bits of $q_1q_2\ldots q_k$ cannot be transmitted yet, because a carry can occur, which can propagate through the entire register containing lower boundary point. Here, two solutions will be investigated that allow the first bits of $Q(.)$ to be transmitted: the Rubin coder, which prevents a carry-over from happening altogether, and the Witten-Neal-Cleary coder, which uses an elegant carry blocking scheme.

3.3.2 SOLUTION I: RUBIN CODER

The Rubin algorithm [35] is an implementation of a binary arithmetic coder with two F bits registers, $P^R(.)$ and $Q^R(.)$. Register $P^R(.)$ is an integer representing the significant bits of $P(.)$. The bits preceding the bits in this register are all zero. Register $Q^R(.)$ is an integer representing the F least significant bits of $Q(.)$. The bits preceding the bits in this register should be fixed. Therefore, the registers have to fulfil the following property, for any $n = 1, 2, \ldots, N$:

$$0 \leq Q^R(x_1^n) < Q^R(x_1^n) + P^R(x_1^n) \leq 2^F. \tag{3.1}$$

The second inequality holds because $P^R(.)$ is a positive integer. The last inequality guarantees that a carry-over will not occur.

The Rubin algorithm realizes bound (3.1) by scaling the two registers only if the first bit of the Q-register is fixed. There are two cases in which the first bit of the Q-register is fixed. First, if at some point $Q^R(x_1^n) + P^R(x_1^n) \leq 2^{F-1}$, then the most significant bit of $Q^R(.)$ is fixed at a zero. Secondly, if at some point $Q^R(x_1^n) \geq 2^{F-1}$, then the most significant bit of $Q^R(.)$ is fixed at a one. Thus, the Rubin algorithm will scale the registers until,

$$Q^R(x_1^n) < 2^{F-1} < Q^R(x_1^n) + P^R(x_1^n). \tag{3.2}$$

holds. As long as bound (3.2) is fulfilled, the most significant bit of $Q^R(.)$ can still take either value.

The encoder is a straightforward implementation of the scaling rule. Before it divides the interval in two subintervals it scales the registers until bound (3.2) is satisfied.

Rubin encoder:

- Initialization. $P^R(\emptyset) = 2^F$, and $Q^R(\emptyset) = 0$.
- For $n = 1, 2, \ldots, N$ do:
  - ⋆ While $Q^R(x_1^{n-1}) < 2^{F-1} < Q^R(x_1^{n-1}) + P^R(x_1^{n-1})$ is *not* fulfilled do:
    - ▷ If $Q^R(x_1^{n-1}) + P^R(x_1^{n-1}) \leq 2^{F-1}$ then:
      - ◇ Transmit a "0",
      - ◇ $Q^R(x_1^{n-1}) = Q^R(x_1^{n-1}) \cdot 2$, and
      - ◇ $P^R(x_1^{n-1}) = P^R(x_1^{n-1}) \cdot 2$,

      else $Q^R(x_1^{n-1}) \geq 2^{F-1}$ and:
      - ◇ Transmit a "1",
      - ◇ $Q^R(x_1^{n-1}) = (Q^R(x_1^{n-1}) - 2^{F-1}) \cdot 2$, and
      - ◇ $P^R(x_1^{n-1}) = P^R(x_1^{n-1}) \cdot 2$.
  - $\overset{(a)}{\star}$ $P^R(x_1^n) = \lfloor P(X_n = x_n | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
    $Q^R(x_1^n) = Q^R(x_1^{n-1}) + \lfloor P(X_n < x_n | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
- Ending. Send $Q^R(x_1^N)$ to the output stream.

Note that the value in register P^R will be as high as 2^F, so this register should have size $F+1$ bits. The rounding in the last step of the loop, marked with (a), should be such that both subintervals have a positive size after the split. Condition (3.2) guarantees that the interval size is at least two, and therefore such a rounding is always possible.

The decoder works quite similar. It uses two extra registers. One is called Q_{REC} (of size F), in which a part of the RECeived code word is stored. The other one is the threshold register T, also of size F.

Rubin decoder:

- Initialization. $P^R(\emptyset) = 2^F$, $Q^R(\emptyset) = 0$, and $Q_{REC}(\emptyset)$ should be filled with the first F symbols from the code word.
- For $n = 1, 2, \ldots, N$ do:
  - ⋆ While $Q^R(x_1^{n-1}) < 2^{F-1} < Q^R(x_1^{n-1}) + P^R(x_1^{n-1})$ is *not* fulfilled do:
    - ▷ If $Q^R(x_1^{n-1}) + P^R(x_1^{n-1}) \leq 2^{F-1}$ then
      - ◇ $Q^R(x_1^{n-1}) = Q^R(x_1^{n-1}) \cdot 2$,

      else $Q^R(x_1^{n-1}) \geq 2^{F-1}$ and:
      - ◇ $Q^R(x_1^{n-1}) = (Q^R(x_1^{n-1}) - 2^{F-1}) \cdot 2$.
    - ▷ $P^R(x_1^{n-1}) = P^R(x_1^{n-1}) \cdot 2$.
    - ▷ $Q_{REC}(x_1^{n-1}) = \big(Q_{REC}(x_1^{n-1}) \& (2^{F-1} - 1)\big) \cdot 2 + C_{NEXT}$, in which & denotes a logical "and"-instruction: this masking operation clears the most significant bit of $Q_{REC}(.)$. C_{NEXT} is the next symbol from the input stream.
  - ⋆ $T(x_1^{n-1}) = Q^R(x_1^{n-1}) + \lfloor P(X_n = 0 | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
  - ⋆ If $Q_{REC}(x_1^n) < T(x_1^{n-1})$ then $x_n = 0$, otherwise $x_n = 1$.
  - ⋆ $P^R(x_1^n) = \lfloor P(X_n = x_n | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
    $Q^R(x_1^n) = Q^R(x_1^{n-1}) + \lfloor P(X_n < x_n | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$

The Rubin algorithm approximates the actual symbol probability by

$$P_{Rubin}(x_n | x_1^{n-1}) = \frac{\lfloor P(x_n | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor}{P^R(x_1^{n-1})}.$$

If $P^R(x_1^{n-1})$ is small, then this approximation will be inaccurate, resulting in some additional redundancy. From bound (3.2) it is clear, that the value of the P^R-register can be as low as 2. Then it will use a symbol probability of a $\frac{1}{2}$, independent of its actual probability. Fortunately, this poor worst case performance almost never happens in practice provided that the Rubin algorithm uses reasonable sized registers ($F \geq 12$).

3.3.3 SOLUTION II: WITTEN-NEAL-CLEARY CODER

The worst case performance of the Rubin algorithm is rather poor. A different, and more modern, arithmetic coder has been proposed by Witten, Neal and Cleary (the WNC coder) [71]. It has a better worst case performance and it can be used for non-binary alphabets too.

Suppose that in a specific step n the interval is in the following range:

$$Q_1 \leq Q^R(x_1^n) < H < Q^R(x_1^n) + P^R(x_1^n) \leq Q_3, \tag{3.3}$$

in which $Q_1 = 2^{F-2}$ marks the point at one quarter of the interval register range, $H = 2^{F-1}$ marks the point at two quarters, and $Q_3 = 3 \cdot 2^{F-2}$ marks the point at three quarters of the range. At

this step the Rubin algorithm cannot rescale the registers yet. But now the next two code word symbols will be either "01" or "10", because the subinterval is entirely in the second and third quarter of the range. So if after the next input symbol, the first new code word symbol is a "0', the following code word symbol will be a "1" and vice versa. With this knowledge, it is possible to scale *at this step*, and remember that after the next code word symbol, another code word symbol will follow with the opposite polarity. If after one scaling step, the registers still satisfy bound (3.3), then the original interval was a subinterval of $[\frac{3}{8}, \frac{5}{8})$ and the next *three* symbols are either "011" or "100". Consequently, the algorithm scales twice and remembers that after the next new code word symbol, two more symbols will follow, with the opposite polarity. Thus these extra scaling steps can be repeated as often as necessary, but the number of extra scaling steps should be counted, e.g. in a variable *count*. An example has been shown in Figure 3.1. After two scaling operations, the encoder determines the next symbol to be "1". Thus it has to add two extra 0's. If the original interval would have been split immediately, it would be a subinterval of the interval $[\frac{4}{8}, \frac{5}{8})$, and thus the first 3 output symbols would be 100 too. But due to these extra scaling steps, the WNC coder will work in general with a higher accuracy than the Rubin coder, which should result in a lower redundancy.

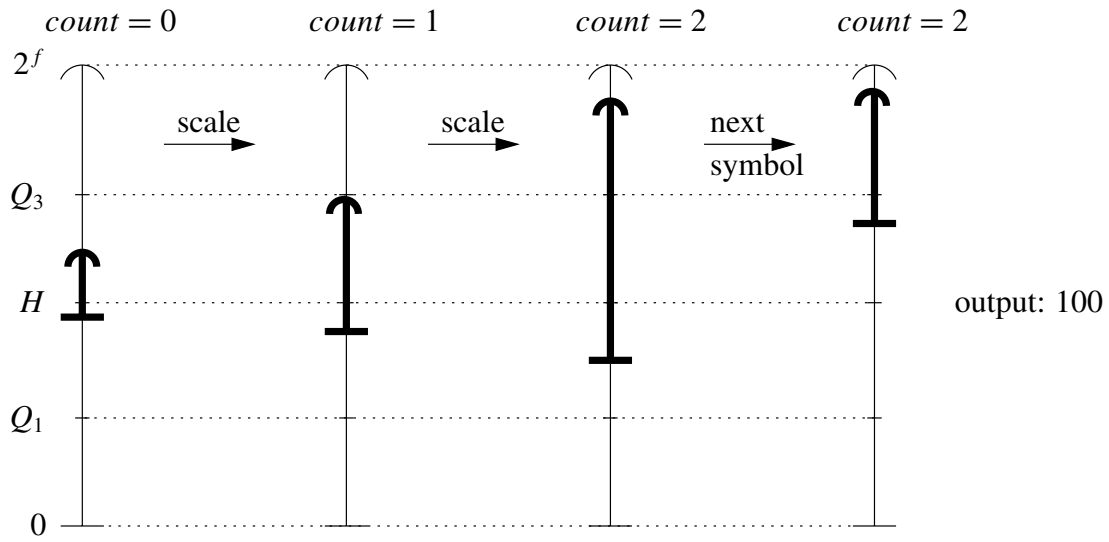


Figure 3.1: An example of the extra scaling steps.

After the scaling operation, registers $P^R(.)$ and $Q^R(.)$ will satisfy at least one of the following two conditions:

$$\begin{aligned} Q^R(x_1^n) < Q_1 < H < Q^R(x_1^n) + P^R(x_1^n), \\ Q^R(x_1^n) < H < Q_3 < Q^R(x_1^n) + P^R(x_1^n). \end{aligned} \tag{3.4}$$

These conditions guarantee that after a scaling step, the interval is at least a quarter of the total interval register range.

The WNC coder differs from the Rubin coder in two ways. First, it has an additional scaling step. The WNC coder first applies the scaling step from Rubin, followed by a scaling step that ensures that the registers also fulfil one of the two conditions in (3.4). Secondly, the WNC coder has a more elegant ending procedure, which is made possible by changing the order of the steps. The WNC coder first splits the interval, and then scales the registers. In this way, after each processed symbol, the new subinterval includes at least the second quarter or the third quarter. In case it includes the second quarter, the encoder can end the encoding by appending zero followed by $count + 1$ ones to the code word. Then, independently of any subsequent code words, this code word represents a point in the subinterval corresponding to the second quarter of the range. In case the third quarter is included in the interval, then it adds a one followed by $count + 1$ zeros to the code word.

WNC encoder:

- Initialization. $P^R(\emptyset) = 2^F$, $Q^R(\emptyset) = 0$, and $count = 0$.
- For $n = 1, 2, \ldots, N$ do:
  - $\star$ $P^R(x_1^n) = \lfloor P(X_n = x_n | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
    $Q^R(x_1^n) = Q^R(x_1^{n-1}) + \lfloor P(X_n < x_n | x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
  - $\star$ (From Rubin) While $Q^R(x_1^n) < 2^{F-1} < Q^R(x_1^n) + P^R(x_1^n)$ does *not* hold do:
    - $\triangleright$ If $Q^R(x_1^n) + P^R(x_1^n) \leq 2^{F-1}$ then
      - $\diamond$ append a "0", followed by $count$ times "1" to the code word,
      - $\diamond$ $Q^R(x_1^n) = Q^R(x_1^n) \cdot 2$,

      else $Q^R(x_1^n) \geq 2^{F-1}$ and:
      - $\diamond$ append a "1", followed by $count$ times "0" to the code word,
      - $\diamond$ $Q^R(x_1^n) = (Q^R(x_1^n) - 2^{F-1}) \cdot 2$.
    - $\triangleright$ $count = 0$, and $P^R(x_1^n) = P^R(x_1^n) \cdot 2$.
  - $\star$ While $Q_1 \leq Q^R(x_1^n)$ and $Q^R(x_1^n) + P^R(x_1^n) \leq Q_3$ do:
    - $\triangleright$ $count = count + 1$,
    - $\triangleright$ $Q^R(x_1^n) = (Q^R(x_1^n) - Q_1) \cdot 2$, and $P^R(x_1^n) = P^R(x_1^n) \cdot 2$.
- Ending. If $Q^R(x_1^n) < Q_1$ then append "0" followed by $count + 1$ times "1" to the code word, otherwise append "1" followed by $count + 1$ times "0".

The decoder is very similar to the encoder. It doesn't need an extra $count$ register, but it needs two other registers: the threshold register T, and the received code word register Q_{REC}. Furthermore, it is possible that the decoder, while decoding the last few symbols, wants to read beyond the end of the code word. In this case the decoder may insert any symbol C_{NEXT} in the Q_{REC}-register.

WNC decoder:

- Initialization. $P^R(\emptyset) = 2^F$, $Q^R(\emptyset) = 0$, and $Q_{REC}(\emptyset)$ should be filled with the first F symbols from the code word.
- For $n = 1, 2, \ldots, N$ do:
  - $T(x_1^{n-1}) = Q^R(x_1^{n-1}) + \lfloor P(X_n = 0|x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
  - If $Q_{REC}(x_1^{n-1}) < T(x_1^{n-1})$ then $x_n = 0$, otherwise $x_n = 1$.
  - $P^R(x_1^n) = \lfloor P(x_n|x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
    $Q^R(x_1^n) = Q^R(x_1^{n-1}) + \lfloor P(X_n < x_n|x_1^{n-1}) \cdot P^R(x_1^{n-1}) \rfloor$
  - (From Rubin) While $Q^R(x_1^n) < 2^{F-1} < Q^R(x_1^n) + P^R(x_1^n)$ does *not* hold do:
    - If $Q^R(x_1^n) + P^R(x_1^n) \leq 2^{F-1}$ then $Q^R(x_1^n) = Q^R(x_1^n) \cdot 2$, else $Q^R(x_1^n) \geq 2^{F-1}$ and $Q^R(x_1^n) = (Q^R(x_1^n) - 2^{F-1}) \cdot 2$.
    - $P^R(x_1^n) = P^R(x_1^n) \cdot 2$
    - $Q_{REC}(x_1^n) = \big(Q_{REC}(x_1^n) \& (2^{F-1} - 1)\big) \cdot 2 + C_{NEXT}$, in which C_{NEXT} is the next symbol from the code word, or a '0' if there are no more code word symbols left.
  - While $Q_1 \leq Q^R(x_1^n)$ and $Q^R(x_1^n) + P^R(x_1^n) \leq Q_3$ do:
    - $Q^R(x_1^n) = (Q^R(x_1^n) - Q_1) \cdot 2$, and $P^R(x_1^n) = P^R(x_1^n) \cdot 2$,
    - $Q_{REC}(x_1^n) = (Q_{REC}(x_1^n) - Q_1) \cdot 2 + C_{NEXT}$.

3.4 IMPLEMENTATION

3.4.1 CHANGING THE FLOATING POINT REPRESENTATION

The probability distribution used in the arithmetic coder is either computed by the CTW algorithm or by the snake algorithm. Both the CTW algorithm and the snake algorithm use a fixed-point log-representation for the probabilities, while the arithmetic coders described here use an integer to store the size of the remaining interval. One solution would be to redesign the arithmetic coder such that it uses the same fixed-point log-representation. Here, an alternative solution is used: the numbers in log-representation will be translated to integers by means of a table. Suppose that the actual estimate of the probability p for symbol x_n is $p = P(X_n = 0|x_1^{N-1})$. This probability will be approximated in the fixed-point log-representation by an integer $\bar{p} = \lfloor 2^f \log_2 p \rceil$ of which the lower f bits are the bits after the decimal point. The table now maps each possible value of $\bar{p}$ to an integer $\tilde{p}$ of length F:

$$\tilde{p} \stackrel{\text{table}}{=} \lfloor 2^{\bar{p}/2^f} 2^F \rceil \approx p 2^F, \tag{3.5}$$

with $0 < \tilde{p} < 2^F$. Special attention has been paid to rounding, while generating this table[2]. Thus the resulting integer $\tilde{p}$ is proportional to the actual probability p. This integer is used to scale the interval size register $P^R(x_1^{n-1})$:

$$P^R(x_1^{n-1}, x_n = 0) = \left\lfloor \frac{P^R(x_1^{n-1})\tilde{p}}{2^F} \right\rceil, \tag{3.6}$$

and

$$P^R(x_1^{n-1}, x_n = 1) = P^R(x_1^{n-1}) - P^R(x_1^{n-1}, x_n = 0). \tag{3.7}$$

Equation (3.6) can be implemented with a single integer multiplication and a right shift over F bit. Equation (3.7) only needs an integer subtraction. Therefore, the entire process of splitting the interval size register $P^R(.)$ given the probabilities in log-representation takes one table access, one integer multiplication, one shift operation and one subtraction.

The number of entries in the table is either limited by the accuracy F used in the arithmetic coder (see (3.5)), or by the accuracy in the CTW algorithm. In turns out that in our case, the limiting factor is the accuracy of CTW. In Chapter 4 we compute that the smallest possible probability used in CTW is $1/4082$ or -3071 in the fixed-point log implementation with $f = 8$. As a result, the table needs to have 3072 entries. The width of each entry in the table depends on the accuracy F, and with $F = 16$ each entries occupies two bytes.

3.4.2 IMPLEMENTING THE RUBIN CODER

The Rubin algorithm is an almost exact implementation of the pseudo code described in Section 3.3.2. $P^R(x_1^n)$ is computed as described by equations (3.6), and (3.7), and with the table described by (3.5). But, it will only compute the new subinterval size for symbol zero, and it will use the remainder of the unsplit interval for symbol one. Furthermore, by adjusting the new subinterval size of symbol zero, it guarantees that both new subintervals have a positive size.

3.4.3 IMPLEMENTING THE WITTEN-NEAL-CLEARY CODER

The Witten-Neal-Cleary coder is implemented slightly different from the pseudo code described in Section 3.3.3. For each source symbol, it first performs the two scaling steps, and then splits the resulting interval size register in two. It uses equations (3.6), and (3.7), and the table described by (3.5) to compute the new subinterval size for symbol zero, and it checks if after the split of the interval register both subintervals will have a positive size. If necessary the subinterval for symbol zero is adjusted. Because the scaling step precedes the splitting step, the encoder has a slightly different ending procedure. If the most significant bit of the lower boundary point register $Q^R(x_1^N)$ is a zero, then the encoder outputs a zero, followed by *count* times a one, followed by the $F - 1$ least significant bits of register $Q^R(x_1^N)$. Similar, if the most significant bit of $Q^R(x_1^N)$ is a one, then it outputs a one followed by *count* times a zero and the $F - 1$ least significant bits

[2]With $\lfloor . \rceil$ we denote an unspecified rounding. But, incorrect rounding might result in an asymmetric arithmetic coder: one that favours sequences with either much more zeros than ones, or vice versa. During evaluation the effect of the rounding needs to be specifically checked.

of register $Q^R(x_1^N)$. We chose this implementation to demonstrate that both coders have the exact same structure. Note that in this way more bits than necessary are needed, although the loss is at most $F - 1$ bits.

3.5 FIRST EVALUATION

The three main criteria for an arithmetic coder are a low complexity, a high compression speed, and a low redundancy. Both arithmetic coders use only a few registers and a small table, thus the memory complexity is very low. As a result our experiments are only concerned with the speed and the performance of the two arithmetic coders. Three different memoryless source sequences of 10,000,000 bits have been created. The parameter of each sequence varies within a small given range. Thus a source symbol is generated by first choosing the probability of a one from the small range, and then by generating the symbol given that probability for a one.

A The probability of a one is in the range [0.45; 0.55]. Now the arithmetic coders will create a compressed sequence of approximately the same length as the source sequence. Because the data compression programs used in our design will never expand the source sequence much, this source sequence will result in the maximum amount of shifts in the arithmetic coders, and thus it is a good test for the lowest compression speed.

B The probability of a one is taken from the interval [0.95; 1]. The arithmetic coders make small rounding errors when they split the interval into two subintervals. The relative influence of these rounding errors can be very high if one subinterval is very small, and the other subinterval is very large. Therefore, this source sequence could result in a high redundancy.

C The probability of a one in the third sequence is in the interval [0.84; 1]. This source sequence tries to simulate the range of probabilities that these arithmetic coders will receive in the final design. If we consider the compression of a text file, then first a Huffman code is applied that will (on average) describe each symbol in five bits. The final design will encode text files to about 2 bits/symbol, thus each of these five bits is on average encoded with about 0.4 bits per bit. This rate corresponds to bits with a probability of about 0.92.

The experiments are performed on our reference test system: a HP 735 workstation.

First the compression speed of the encoders is measured. The arithmetic coders can use accuracies of up to 16 bits. In order to measure the influence of the accuracy on the compression speed, the speed has been measured for an accuracy of 4 bits (Table 3.1) and 16 bits (Table 3.2). As expected sequence A results in the lowest compression speed. The WNC encoder is slower than the Rubin encoder, since it has to perform an additional scaling step, but the difference is less than 10 %. Also, the accuracy has only a very small influence on the compression speed: increasing the accuracy from 4 bits to 16 bits results in a loss in compression speed of less than 5 %. This was expected, since the arithmetic coder only uses 16 bit integer operations and as long as the integers fit in the registers of the micro-processor, there will be no penalty in

computational speed. Our reference system has a 32 bits micro-processor. On each sequence and with each accuracy, both arithmetic coders compress faster than 850 kbits/second, and therefore they both satisfy the speed criterion. Note that this also includes the very small overhead of the main loop of the measuring program.

Table 3.1: Speed of the arithmetic encoders with $F = 4$.

	Rubin		WNC	
	sec.	kbits/sec.	sec.	kbits/sec.
A	10.43	936.3	11.07	882.2
B	7.17	1362.0	7.76	1258.5
C	7.95	1228.4	8.62	1132.9

Table 3.2: Speed of the arithmetic encoders with $F = 16$.

	Rubin		WNC	
	sec.	kbits/sec.	sec.	kbits/sec.
A	10.82	902.6	11.43	854.4
B	7.44	1312.6	7.95	1228.4
C	8.32	1173.8	8.87	1101.0

Next, the redundancy of these two encoders has to be measured. The measurement program computes the new probability distribution as a set of floating point numbers, which have to be transformed to fixed-point log numbers first. This induces a "translation" redundancy, which is computed by the measurement program too. The translation redundancy varies between 1 byte (sequence A) and 71 bytes (sequence B).

Table 3.3 summarizes the redundancies of the Rubin coder, Table 3.4 for the WNC coder. Both the absolute redundancies in bytes as the relative redundancies in bits/symbol are given for four different accuracies. The relative redundancy is computed by dividing the redundancy in bits by the number of input bytes. Note that in the final design the source symbols are coded in five bits on average. By computing the redundancy here per 8 input bits we are only tightening the requirement. Two additional test sequences B* and C*, which are the inverse sequences of B and C, have been added to the test, in order to check for asymmetrical behaviour of the arithmetic coders.

As expected, especially for the lower accuracies, sequence B results in general in the highest redundancy. For higher accuracies (12 bits and more for the WNC coder, 16 bits for the Rubin coder), the coders work so well that the redundancy is almost zero. At almost every accuracy and for every test sequence the WNC coder performs better than the Rubin coder. The WNC coder already achieves a redundancy below 0.01 bits/symbol with eight bits accuracy, the Rubin coder

Table 3.3: The absolute and relative redundancy of the Rubin encoder.

	$F = 4$		$F = 8$		$F = 12$		$F = 16$	
	bytes	bits/sym.	bytes	bits/sym.	bytes	bits/sym.	bytes	bits/sym.
A	24927	0.1595	2177	0.0139	144	0.0009	9	0.0001
B	62250	0.3984	3983	0.0255	389	0.0025	23	0.0001
B*	62249	0.3984	4745	0.0304	396	0.0025	19	0.0001
C	29912	0.1914	4479	0.0287	327	0.0021	28	0.0002
C*	57613	0.3687	4968	0.0318	377	0.0024	23	0.0001

Table 3.4: The absolute and relative redundancy of the WNC encoder.

	$F = 4$		$F = 8$		$F = 12$		$F = 16$	
	bytes	bits/sym.	bytes	bits/sym.	bytes	bits/sym.	bytes	bits/sym.
A	11180	0.0716	137	0.0009	3	0.0000	3	0.0000
B	62250	0.3984	1509	0.0097	-1	-0.0000	1	0.0000
B*	62249	0.3984	1542	0.0099	4	0.0000	-9	-0.0001
C	29912	0.1914	765	0.0049	1	0.0000	2	0.0000
C*	46500	0.2976	815	0.0052	7	0.0000	0	0.0000

requires a higher accuracy. At 12 bits accuracy and especially at 16 bits, both encoders easily achieve the performance criterion.

Both arithmetic coders fulfil all three criteria. Based on its superior performance, at the cost of only a very slight increase in complexity, we chose to use the WNC encoder with 16 bits accuracy.

4

SUBSYSTEM 2: THE CONTEXT-TREE WEIGHTING ALGORITHM

This chapter discusses the implementation of a modelling algorithm for the final design, based on the CTW algorithm. The CTW algorithm is described first, followed by a modification, the binary decomposition, which extends the CTW algorithm for non-binary sources. An implementation of the CTW algorithm has to perform two actions: computing and searching. The computations will be performed in a fixed-point log-domain and several steps will be replaced by simple table look-ups. Searching is implemented by means of an incremental search technique in a hash table, which combines small records with a high searching speed. Finally, the parameters of the CTW algorithm are fixed and the resulting data compression system is evaluated. It runs in 32 MBytes, but it is too slow, and although it achieves the performance requirement on text files, its performance on other files is insufficient.

4.1 DESCRIPTION

THIS chapter is concerned with the design of the CTW algorithm. This is a sequential modelling algorithm, and it should provide estimates of the probabilities of the subsequent source symbols to the arithmetic coder. The final implementation has to fulfil the following requirements:

- The final design should, together with the arithmetic coder, achieve a performance on the Calgary corpus comparable to two competitive algorithms, the "state selection" algorithm

and PPMZ-2. On text files it should lose not more than 0.02 bits/symbol, on the other files it should not lose more than 0.04 bits/symbol.

- The final design may not use more than 32 MBytes of memory.
- The final design, together with the arithmetic coder, should achieve a compression speed of at least 10 kBytes/second.

All choices that lead to our implementation of the CTW algorithm will be discussed in this chapter. But first, the CTW algorithm itself will be explained, since only a thorough understanding of the algorithm can result in an efficient design.

4.2 BASIC ALGORITHM

The basic CTW algorithm is a modelling algorithm for binary tree sources. This section first explains some necessary concepts, e.g. binary tree sources, before it discusses the CTW algorithm and a first basic implementation.

4.2.1 TREE SOURCES

Chapter 1 discussed a universal source coding algorithm for memoryless sources. In general the source sequences that need to be compressed (like texts, and source codes), are generated by sources with memory. Here we will assume that the source is a tree source. A **tree source** is described by a suffix set $\mathcal{S}$ and a parameter set $\Theta_{\mathcal{S}}$. The suffix set should be such that no string in $\mathcal{S}$ is equal to the suffix from an other string in $\mathcal{S}$, and each possible semi-infinite string should have a suffix in $\mathcal{S}$. The parameter set $\Theta_{\mathcal{S}}$ contains exactly one parameter θ_s for each string s in $\mathcal{S}$. Parameter θ_s is the probability that the tree source will generate a one, given that the suffix of the source sequence so far is s. We will assume that the longest suffix in $\Theta_{\mathcal{S}}$ is finite, and has at most D symbols. This implies that in sequences generated by a tree source, the D symbols $x_{n-D}, \ldots, x_{n-1}$ preceding a symbol x_n determine the parameter with which x_n was generated. These symbols are called the **context** of x_n.

Example: Consider a tree source $\mathcal{S} = \{00, 10, 1\}$ (visualized in Figure 4.1). Symbol x_{10} has context $\ldots 00$, thus parameter θ_{00} was used to generate it. Symbols x_{11}, x_{12} and x_{14} all have context $\ldots 1$, therefore parameter θ_1 was used to generate each of them. Finally, the new symbol x_{15} has context $\ldots 10$ (just like symbol x_{13}), it will be a one with probability θ_{10}.

4.2.2 KNOWN TREE STRUCTURE WITH UNKNOWN PARAMETERS

Suppose that the model $\mathcal{S}$ of the tree source that generated a source sequence is known, but its parameters $\Theta_{\mathcal{S}}$ are unknown. Is it still possible to compress this source sequence?

All symbols following the same context s will be generated by the same parameter θ_s. Thus these symbols form a memoryless subsequence within the source sequence. Therefore, each

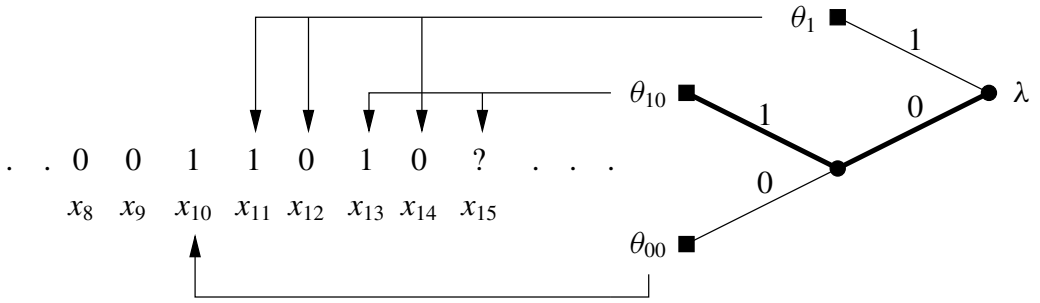


Figure 4.1: An example of a tree source.

context s in the set $\mathcal{S}$ corresponds to a, possibly empty, disjoint memoryless subsequence and all these subsequences together form the entire source sequence.

Example (continued): The model of the tree source is $\mathcal{S} = \{00, 10, 1\}$ and the parameter vector is $\Theta_{\mathcal{S}} = \{\theta_{00}, \theta_{10}, \theta_1\}$. The tree has three leaves. Thus the source sequence can be split into three disjoint memoryless subsequences, one for each leaf (see Figure 4.2).

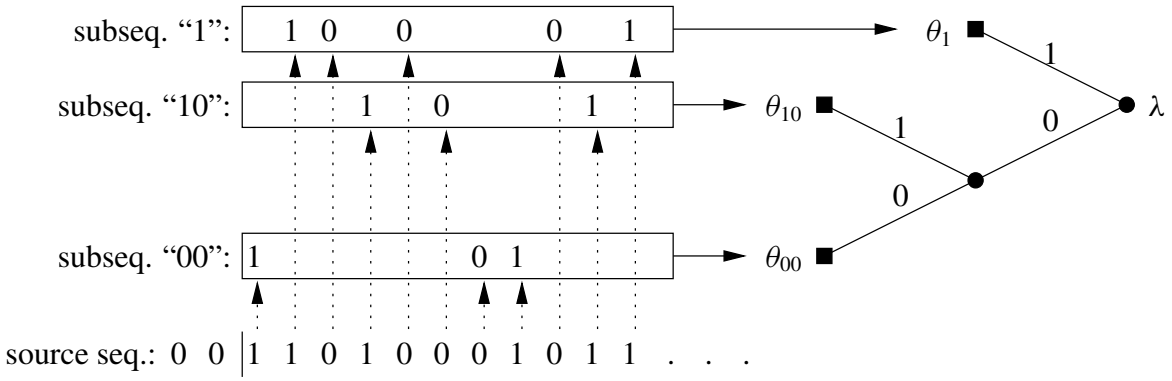


Figure 4.2: Splitting a source sequence in memoryless subsequences.

The KT-estimator (see Section 1.5.6) can be used to code a memoryless source sequence. It uses the number of zeros a and the number of ones b in the source sequence so far to estimate the probability distribution of the next symbol. Similarly, it is possible to code the source sequence in the case of tree sources. Now, each leaf s of the tree source has to have its own two counters a_s and b_s, counting the number of zeros and ones respectively that followed context s in the source sequence so far. If the new source symbol has context s, then its estimated probability can be found by applying the KT-estimator with counts a_s and b_s. Next, either a_s or b_s is incremented, depending on the value of the new symbol. With this procedure the estimated probability of the entire memoryless subsequence corresponding to a context s is $P_e(a_s, b_s)$, if a_s zeros and b_s ones followed context s. The probability assigned to the entire source sequence x_1^N, is the product of the estimated probabilities of each disjoint memoryless subsequence: $P_e(x_1^N|\mathcal{S}) = \prod_{s\in\mathcal{S}} P_e(a_s, b_s)$. The following theorem gives an upper bound on the individual redundancy of this algorithm.

Theorem 4.1 (Redundancy in case of a known tree source) *Consider a binary source sequence x_1^N generated by a tree source with known model $\mathcal{S}$, with $|\mathcal{S}|$ leaves. The estimated probabilities of the modelling algorithm described above can be used as coding probabilities in an arithmetic coder. Such an algorithm will achieve a code word length on the source sequence for which the following upper bound on the individual cumulative redundancy holds (for $N \geq |\mathcal{S}|$):*

$$\rho(x_1^N) \stackrel{\triangle}{=} l(x_1^N) - \log_2 \frac{1}{P_a(x_1^N)} \leq \left(\frac{|\mathcal{S}|}{2} \log_2 \frac{N}{|\mathcal{S}|} + |\mathcal{S}| \right) + 2. \tag{4.1}$$

Proof: The individual redundancy in (4.1) is the sum of two redundancies:

$$l(x_1^N) - \log_2 \frac{1}{P_a(x_1^N)} = \left(\log_2 \frac{1}{\prod_{s \in \mathcal{S}} P_e(a_s, b_s)} - \log_2 \frac{1}{P_a(x_1^N)} \right) + \left(l(x_1^N) - \log_2 \frac{1}{\prod_{s \in \mathcal{S}} P_e(a_s, b_s)} \right).$$

The first term is the parameter redundancy and the second term is the coding redundancy.

The parameter redundancy is caused by the estimators. The source has $|\mathcal{S}|$ leaves. If the subsequence corresponding to context s has $N_s = a_s + b_s \geq 1$ counts, then the redundancy of the KT-estimator on the corresponding sequence is less than $\frac{1}{2}\log_2(N_s) + 1$ bits. A leaf s without counts ($N_s = 0$) does not contribute to the redundancy. In this proof we need a concave function describing an upper bound on the redundancy for all $N_s \geq 0$. The following function handles this in an elegant way:

$$\gamma(z) = \begin{cases} z, & \text{for } 0 \leq z < 1, \\ \frac{1}{2}\log_2 z + 1, & \text{for } z \geq 1. \end{cases} \tag{4.2}$$

Jensen's inequality [16] states that for concave functions f (the straight line that connects two arbitrary points on the curve lies completely on or below the curve) and a random variable X, the following inequality holds,

$$Ef(X) \leq f(EX). \tag{4.3}$$

The actual probability of this sequence is $P_a(x_1^N) = \prod_{s \in \mathcal{S}} (1-\theta_s)^{a_s} \theta_s^{b_s}$, in which leaf s has parameter θ_s. The parameter redundancy of the $|\mathcal{S}|$ estimators for any source sequence x_1^N can be upper bounded by:

$$\begin{aligned} \log_2 \frac{1}{P_e(x_1^N|\mathcal{S})} - \log_2 \frac{1}{P_a(x_1^N)} &\stackrel{(a)}{=} \sum_{s \in \mathcal{S}, a_s + b_s > 0} \left(\log_2 \frac{1}{P_e(a_s, b_s)} - \log_2 \frac{1}{(1-\theta_s)^{a_s} \theta_s^{b_s}} \right) \\ &\stackrel{(b)}{\leq} \sum_{s \in \mathcal{S}, a_s + b_s > 0} \frac{1}{2}\log_2(N_s) + 1 = \sum_{s \in \mathcal{S}} \gamma(N_s) = |\mathcal{S}| \sum_{s \in \mathcal{S}} \frac{1}{|\mathcal{S}|} \cdot \gamma(N_s) \\ &\stackrel{(c)}{\leq} |\mathcal{S}| \gamma \left(\sum_{s \in \mathcal{S}} \frac{N_s}{|\mathcal{S}|} \right) = |\mathcal{S}| \gamma \left(\frac{N}{|\mathcal{S}|} \right) \\ &\stackrel{(d)}{=} \frac{|\mathcal{S}|}{2} \log_2 \frac{N}{|\mathcal{S}|} + |\mathcal{S}|, \end{aligned}$$

for $N \geq |\mathcal{S}|$. In (a) the redundancy for the entire source sequence is split in a sum of the redundancies of the memoryless subsequences. Inequality (b) follows from the parameter redundancy of one KT-estimator (1.11) and inequality (c) follows from Jensen's inequality (4.3) with the γ-function. Equality (d) holds only for $N \geq |\mathcal{S}|$.

The individual coding redundancy is caused by the arithmetic code and is less than two bits (see Theorem 1.4). The sum of these two redundancies satisfies the theorem. □

4.2.3 WEIGHTING PROBABILITY DISTRIBUTIONS

Suppose it is known that the source sequence was generated by one of two known models $\mathcal{S}_1$ and $\mathcal{S}_2$. The modelling algorithm now has to find a good coding distribution over the source sequences for both models. It can use the weighted distribution. For all $n = 1, 2, \ldots, N$, the weighted probability of source sequence x_1^n is defined as:

$$P_w(x_1^n) = \frac{P(x_1^n|\mathcal{S}_1) + P(x_1^n|\mathcal{S}_2)}{2}.$$

The sum of the weighted probabilities over all possible source sequences x_1^n is equal to one. Furthermore, this probability distribution also has the property that $P_w(x_1^{n-1}) = P_w(x_1^{n-1}, X_n = 0) + P_w(x_1^{n-1}, X_n = 1)$. Therefore, it can be used as the coding distribution in an arithmetic coder. Since the arithmetic coders described in Chapter 3 use conditional coding probabilities, Bayes' rule is used to compute the conditional weighted probabilities:

$$P_w(x_n|x_1^{n-1}) = \frac{P_w(x_1^n)}{P_w(x_1^{n-1})}.$$

Clearly, the product of these conditional weighted probabilities results in $\prod_{n=1}^{N} P_w(x_n|x_1^{n-1}) = P_w(x_1^N)$. A data compression algorithm that uses the weighted probability distribution as the coding distribution for an arithmetic coder to code a source sequence x_1^N achieves a code word length $l(x_1^N)$:

$$\begin{aligned} l(x_1^N) &\overset{(a)}{<} \log_2 \frac{1}{P_w(x_1^N)} + 2 = \log_2 \frac{2}{P(x_1^N|\mathcal{S}_1) + P(x_1^N|\mathcal{S}_2)} + 2 \\ &\leq \log_2 \frac{2}{P(x_1^N|\mathcal{S})} + 2 = \left(\log_2 \frac{1}{P(x_1^N|\mathcal{S})} + 1 \right) + 2, \end{aligned}$$

in which $\mathcal{S}$ is $\mathcal{S}_1$ or $\mathcal{S}_2$: the model used to generate the source sequence. Inequality (a) holds because the weighted distribution $P_w(x_1^N)$ is used as coding distribution in an optimal arithmetic coder (Theorem 1.4). The cumulative redundancy increases by not more than one bit. This additional redundancy is called model redundancy and is due to the uncertainty about the actual model. Note that an alternative algorithm, which first computes both $P(x_1^N|\mathcal{S}_1)$ and $P(x_1^N|\mathcal{S}_2)$, and then uses the model that resulted in the highest probability, needs exactly one bit to specify the best model to the decoder. Thus an algorithm using the weighted probability will not perform worse than this alternative algorithm. A more in depth discussion on weighting can be found in Appendix A.

4.2.4 THE CTW ALGORITHM

The context-tree weighting algorithm [62] is a modelling algorithm that assumes that the source is a tree source. It uses a context tree to store all necessary counters. A context tree is a full suffix tree of depth D, thus a binary context tree has 2^D leaves. Every node and leaf s in the context tree has two counters a_s and b_s, which count the number of zeros and ones respectively that followed context s in the source sequence so far. A context tree can be easily updated. Suppose it is already updated for source symbols x_1^{n-1}. Then the context tree for symbols x_1^n can be found by incrementing the count corresponding to the value of x_n in the D nodes and in the leaf on the path between root node λ and the leaf corresponding to $x_{n-D}, \ldots, x_{n-1}$. This path in the context tree is called the *context path* of symbol x_n. For an example see Figure 4.3.

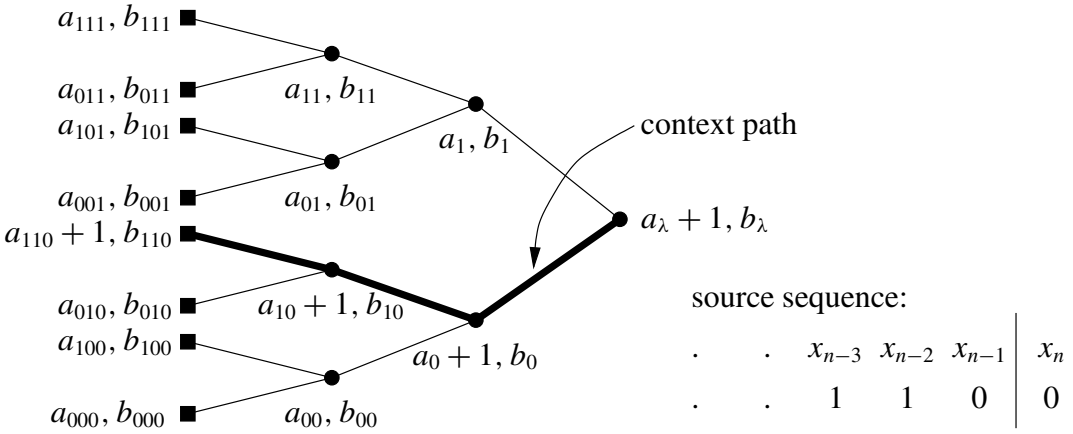


Figure 4.3: Example of an update of the context tree.

For every possible tree source $\mathcal{S}$ with depth smaller than or equal to D, the context tree contains the counts for each of its leaves. Because the CTW algorithm does not know the tree structure $\mathcal{S}$ in advance, it will weight over all possible tree structures, which includes $\mathcal{S}$. The beauty of the CTW algorithm is the efficient arithmetic used to weight over all tree structures. The weighted probability is computed recursively, from the leaves of the context tree, downwards to the root node. In the leaf and in each node it computes the following probability:

$$P_w^s(x_1^n) = \begin{cases} P_e(a_s, b_s), & \text{if } |s| = D, \\ \frac{1}{2}P_e(a_s, b_s) + \frac{1}{2}P_w^{0s}(x_1^n)P_w^{1s}(x_1^n), & \text{if } |s| < D, \end{cases} \tag{4.4}$$

in which $|s|$ denotes the length of context s. In leaves s of the context tree, the CTW algorithm has to assume that the symbols following context s form a memoryless subsequence, therefore it uses the KT-estimator to estimate the probability. In nodes s of the context tree, the CTW algorithm weights between the two possibilities. If s is a leaf of the actual model, the symbols following context s form a memoryless subsequence again, and the CTW algorithm should use the KT-estimator to estimate the probability. If s is an internal node of the actual model, the next context symbol is used to split the subsequence corresponding to context s into two subsequences, one corresponding to context $0s$ and one to context $1s$. As a result, the probability of the subsequence

in node s is the product of the probabilities of both subsequences. Because the CTW algorithm has no a priori knowledge about the model, each alternative gets weight a half.

The weighted probability found in the root node of the context tree, $P_w^\lambda(x_1^n)$, is a weighted probability over the probabilities $P(x_1^n|\mathcal{S})$ that all possible tree structures $\mathcal{S}$ assign to the source sequence so far: $P_w^\lambda(x_1^n) = \sum_{\mathcal{S}} w_{\mathcal{S}} P(x_1^n|\mathcal{S})$. From (4.4) it is clear that for every leaf s of the actual model $P_w^s(x_1^N) \geq \frac{1}{2} P_e^s(a_s, b_s)$ holds, and for every node s, $P_w^s(x_1^N) \geq \frac{1}{2} P_w^{0s}(x_1^N) P_w^{1s}(x_1^N)$. Thus, compared to an algorithm that knows the actual tree structure $\mathcal{S}$, the CTW algorithm loses at most a factor a $\frac{1}{2}$ in every node and leaf of the actual tree structure. The actual tree structure has $|\mathcal{S}|$ leaves and $|\mathcal{S}| - 1$ nodes, therefore the weight $w_{\mathcal{S}}$ assigned to tree structure $\mathcal{S}$ is at least $w_{\mathcal{S}} \geq 2^{1-2|\mathcal{S}|}$, and the CTW algorithm loses at most $2|\mathcal{S}| - 1$ bits. Note that if only the encoder knows the actual tree structure, and it has to specify it to the decoder, it will also need about 1 bit per node and leaf.

Theorem 4.2 (Redundancy of the CTW algorithm) *Consider a binary source sequence x_1^N generated by an unknown tree source $\mathcal{S}$, with $|\mathcal{S}|$ leaves. For any tree source $\mathcal{S}$ and any parameter vector $\Theta_{\mathcal{S}}$ the individual redundancy of the CTW algorithm combined with the Elias algorithm is upper bounded by (given that $N \geq |\mathcal{S}|$):*

$$\begin{aligned}
\rho(x_1^n) &= l(x_1^N) - \log_2 \frac{1}{P_a(x_1^N)} \\
&= \log_2 \frac{\prod_{s\in\mathcal{S}} P_e(a_s, b_s)}{P_w^\lambda(x_1^N)} + \log_2 \frac{P_a(x_1^N)}{\prod_{s\in\mathcal{S}} P_e(a_s, b_s)} + \left(l(x_1^N) - \log_2 \frac{1}{P_w^\lambda(x_1^N)} \right) \\
&\leq (2|\mathcal{S}| - 1) + \left(\frac{|\mathcal{S}|}{2} \log_2 \frac{N}{|\mathcal{S}|} + |\mathcal{S}| \right) + 2, \qquad (4.5)
\end{aligned}$$

in which $P_a(x_1^N)$ is the actual probability. The first term is the model redundancy, the second term is the parameter redundancy of a known tree source $\mathcal{S}$ (see Theorem 4.1), and the last term is the coding redundancy (see Theorem 1.4). An extensive proof can be found in [62].

Example: Let us illustrate the performance of the CTW algorithm with an example. Suppose the source sequence x_1^N was generated by a tree source with three leaves: $\mathcal{S} = \{00, 10, 1\}$. We use a context tree with depth $D = 3$. The weighted probability in the root node of the context tree now satisfies:

$$\begin{aligned}
P_w^\lambda(x_1^N) &= \frac{1}{2} P_e(a_\lambda, b_\lambda) + \frac{1}{2} P_w^0(x_1^N) P_w^1(x_1^N) \geq \frac{1}{2} P_w^0(x_1^N) P_w^1(x_1^N) \\
&= \frac{1}{2} \left(\frac{1}{2} P_e(a_0, b_0) + \frac{1}{2} P_w^{00}(x_1^N) P_w^{10}(x_1^N) \right) \left(\frac{1}{2} P_e(a_1, b_1) + \frac{1}{2} P_w^{01}(x_1^N) P_w^{11}(x_1^N) \right) \\
&\geq \frac{1}{2} \cdot \left(\frac{1}{2} P_w^{00}(x_1^N) P_w^{10}(x_1^N) \right) \cdot \left(\frac{1}{2} P_e(a_1, b_1) \right) \\
&\geq \frac{1}{2} \left(\frac{1}{2} \cdot \frac{1}{2} P_e^{00}(a_{00}, b_{00}) \cdot \frac{1}{2} P_e^{10}(a_{10}, b_{10}) \right) \cdot \left(\frac{1}{2} P_e(a_1, b_1) \right) \\
&= 2^{-5} \prod_{s\in\{00,10,1\}} P_e(a_s, b_s).
\end{aligned}$$

The actual tree source has three leaves, and two nodes. Indeed, the weight assigned to tree structure $\mathcal{S}$ is $w_{\mathcal{S}} = 2^{-5}$. As a result, the weighted probability is at most a factor 2^{-5} smaller than the probability assigned to this sequence by an algorithm that knows the actual tree structure $\mathcal{S}$, thus the model redundancy is smaller than 5 bits.

4.2.5 THE BASIC IMPLEMENTATION

The basic implementation stores in each node and leaf s of the context tree not only the counts a_s and b_s, but also the estimated probability $P_e^s(a_s, b_s)$ and in the nodes the weighted probability $P_w^s(x_1^n)$. The computational complexity of the CTW algorithm is linear in the length of the source sequence, because after processing a source symbol, only the counts and probabilities along the context path have to be updated. The new estimated probability in each node and leaf s on the context path can be easily computed sequentially:

$$P_e^s(a_s + 1, b_s) = P_e^s(a_s, b_s) \cdot \frac{a_s + \frac{1}{2}}{a_s + b_s + 1}, \tag{4.6}$$

if the new symbol is a '0' and similar in case it is a '1'. The new weighted probability in each node and leaf on the context path can be computed with (4.4). This formula can be applied directly, because the iterative update procedure starts in the leaf of the context path, and as a result the weighted probabilities in the children of a node are already available and up-to-date, the moment the node itself is being updated.

The following steps describe an implementation of the basic CTW encoding algorithm. The CTW algorithm uses D context symbols, but the first D symbols of the source sequence do not have D context symbols. This can be solved by sending the first D symbols of the source sequence unencoded.

- Initialization: Construct an empty context tree of depth D in which all counts are set to zero, and all probabilities to one. Send the first D symbols uncoded.
- For all $n = D + 1, D + 2, \ldots, N$ do:
  - ⋆ For all $d = D, D - 1, \ldots, 0$ and $s = x_{n-d}, x_{n-d+1}, \ldots, x_{n-1}$ do:
    - ▷ Compute $P_e^s(a_s + 1, b_s) = P_e^s(a_s, b_s) \cdot \frac{a_s + \frac{1}{2}}{a_s + b_s + 1}$ if x_n is a zero, and similar if $x_n = 1$.
    - ▷ Compute $P_w^s(x_1^n) = \frac{1}{2} P_e(a_s, b_s) + \frac{1}{2} P_w^{0s}(x_1^n) P_w^{1s}(x_1^n)$ if $d < D$, or $P_w^s(x_1^n) = P_e(a_s, b_s)$ if $d = D$.
    - ▷ Update the count corresponding to x_n.
  - ⋆ Compute conditional probability $P_w^\lambda(x_n | x_1^{n-1}) = \frac{P_w^\lambda(x_1^n)}{P_w^\lambda(x_1^{n-1})}$ for the arithmetic coder.

In the following sections it will be explained that in practice both $P_w^\lambda(0|x_1^{n-1})$ and $P_w^\lambda(1|x_1^{n-1})$ are being computed for the arithmetic coder instead of just $P_w^\lambda(x_n|x_1^{n-1})$.

The decoding algorithm is similar but is split into two parts: first the weighted probability is computed, and only after the arithmetic decoder decoded the new source symbol, the counts and probabilities can be updated.

4.3 ALGORITHMIC SOLUTIONS FOR THE INITIAL PROBLEMS

4.3.1 BINARY DECOMPOSITION

The CTW algorithm as discussed in the previous section works for binary source sequences only. Data files on computer systems often use bytes to store the information. Because a byte is just 8 bits, it is trivial to use the binary CTW algorithm as modelling algorithm for all data files. But in general each bit of a byte has a special meaning, and ignoring the position of the bits in the bytes will result in a very poor performance. There are two solutions for this problem. First, a new CTW algorithm can be designed that operates on bytes. Secondly, the bytes can be split into bits in such a way that the position of each bit in a byte is kept, while at the same time the binary CTW algorithm can be used.

SOLUTION 1: A 256-ARY CTW ALGORITHM

A data compression program that uses the CTW algorithm for bytes differs in three ways from one that uses the binary CTW algorithm. First, the arithmetic coder now has to handle 256-ary probability distributions. The WNC coder, discussed in Section 3.3.3 can be designed to handle 256-ary alphabets. The other two modifications concern the CTW algorithm. First, the context symbols are now bytes instead of bits and each node in the context tree has now up to 256 children instead of two. Therefore, the weighted probability for nodes s should now be computed as:

$$P_w^s(x_1^n) = \frac{1}{2}P_e(a_s, b_s) + \frac{1}{2}\prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n), \quad \text{for } |s| < D, \tag{4.7}$$

instead of (4.4). Also, the weights $(\frac{1}{2}, \frac{1}{2})$ might be changed. Secondly, an estimator for the non-binary alphabet has to be used. The KT-estimator for an alphabet $\mathcal{X}$, with $|\mathcal{X}|$ symbols assigns the following probability to a source sequence x_1^N, which has a_t symbols t:

$$P_e^s(a_1, a_2, \ldots, a_{|\mathcal{X}|}) = \frac{\prod_{t\in\mathcal{X}}(a_t - \frac{1}{2})!}{(N - 1 + \frac{|\mathcal{X}|}{2})!}.$$

For binary sequences this probability reduces to the binary KT-estimator discussed in Section 1.5.6. For 256-ary symbols the constant term in the denominator is +127, because the estimator has to reserve code space for all 256 possible symbols. In practice most contexts are often followed by a small subset of all possible source symbols, hence a lot of code space is lost and the estimator has a poor performance. One solution for this "small subalphabet"-problem is the escape mechanism. The alphabet is enlarged by an extra symbol, the escape symbol. The estimator will now give a positive probability to the symbols that already occurred, and to the escape symbol. If a new symbol occurs, the escape symbol is coded, followed by coding the new symbol. The new symbol is then added to the alphabet of the estimator. This escape mechanism is used in the PPM algorithm [7, 13] and also in the CTW algorithm [2]. As an alternative solution, the estimator can be a weighting over multiple estimators, each working on a different subalphabet [46].

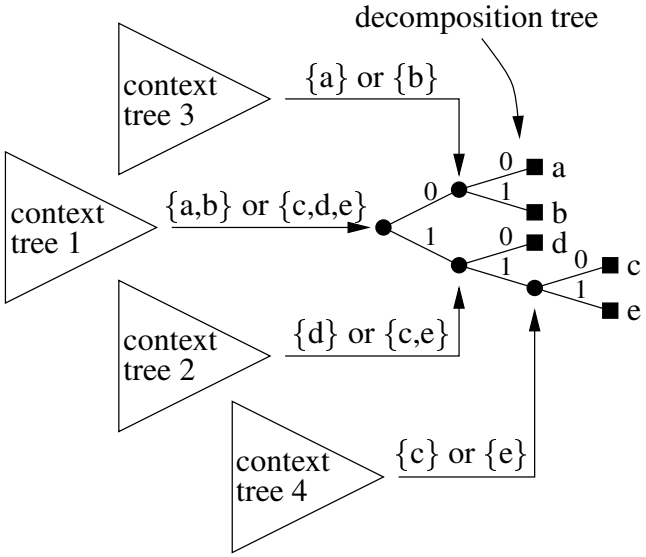


Figure 4.4: An example of a binary decomposition

SOLUTION 2: A BINARY DECOMPOSITION

The other technique to modify the CTW algorithm for non-binary data is by using a binary decomposition. It requires two modifications. First, the non-binary source symbols are mapped to series of binary decisions by means of a decomposition tree [47]. A decomposition tree is a binary tree with one leaf for each possible source symbol. A source symbol is defined by describing the path in the decomposition tree from the root node to the leaf corresponding to that symbol. In each node on this path a binary decision, a bit of the symbol, has to be coded. To code these bits, each node in the decomposition tree has its own context tree, which is used by the CTW algorithm to code only this particular decision. Secondly, the context symbols are non-binary. The representation of the context symbols is completely independent from the binary decomposition tree. Therefore, the context symbols can be decomposed into bits, nibbles, or any other format. This is called the backward decomposition. It has been investigated in [48], and indeed a small gain in performance was achieved by developing specific decompositions for the context symbols. We decided not to decompose the context symbols, and therefore each node in the context trees can have up to 256 children and the CTW algorithm should use (4.7) to compute the weighted probability.

Example: Consider the decomposition tree for a 5-ary source with alphabet $\mathcal{X} = \{a, b, c, d, e\}$ in Figure 4.4. Suppose the new source symbol is a "c". The root node decides whether the new source symbol is in the set $\{a,b\}$ or in the set $\{c,d,e\}$. These decisions are counted in context tree 1. The CTW algorithm uses this context tree to encode a "1". Then the next node is visited, and the CTW algorithm uses context tree 2 to code that decision, a "1". Finally, in the last node on the path, context tree 4 is used by the CTW algorithm to encode a "0". This completes the encoding of the entire path from the root node to the leaf.

OUR DECISION

Here we chose to use the binary decomposition. It has advantages over each of the alternatives[1]. The CTW algorithm with the 256-ary KT-estimator has a very poor performance compared to the CTW algorithm with the binary decomposition, because in practice many leaves will use a subalphabet which consists of just a few symbols. The multi-alphabet CTW algorithm that uses the estimator that weights over different subalphabets, is very computationally intensive, since the computation of the estimator is rather complex[2]. Therefore, the only realistic alternative is the multi-alphabet CTW algorithm that uses an estimator with an escape mechanism. But also this alternative has some disadvantages. First of all, finding an algorithm with a solid scientific basis to compute the escape mechanism is very complicated, and as a result these are often empirically determined. Secondly, the implementation of this multi-alphabet algorithm is more complicated because it uses a 256-ary symbol alphabet, and the arithmetic coder will need to have both the probability of the new source symbol, and the cumulative probability of all "lower" source symbols. Finally, the binary decomposition makes a different trade-off between model cost and parameter cost than do multi-alphabet algorithms, which can result in a superior performance. A CTW algorithm with binary decomposition converges to a model in each context tree separately, therefore these models can differ. As a result, if a certain context is being used as a leaf of the model in one context tree, that same context can be used as an internal node in an other context tree, while in a third context tree the model might be using an even shorter context. This flexibility results in a very efficient use of parameters: longer contexts and more parameters are only used at those places in the context trees in which it is profitable. Thus, the binary decomposition trades additional model cost, because the model in each of the 255 context trees contributes to the model cost, for lower parameter cost.

A drawback of the binary decomposition can be the higher memory complexity, since it uses 255 context trees instead of one. But this will be countered by some efficient pruning algorithms, that will be described in Section 4.4.2.

One open question remains. Given a set of source symbols, many different decomposition trees are possible. Each tree codes the source symbols in a different way. It is not clear which decomposition tree will result in the best performance, or in the lowest complexity. The most straightforward decomposition uses the binary representation of each symbol as its decomposition. In that case, for symbols consisting of 8 bits, we get a decomposition tree with 256 leaves, each at depth 8. We call this decomposition the ASCII decomposition. Different decompositions are described in Chapter 5.

[1] Very interesting is another alternative that has been developed after the completion of this work. It is a CTW algorithm with a binary decomposition, but with an additional weighting over the possible symbol alphabet decompositions [68]. In this way the algorithm can use in each node the best alphabet size: in some nodes it might estimate the probability of the next nibble, while in an other node it uses single bits.

[2] Although more recent investigations [60] show that it might be possible to compute the subalphabet estimator efficiently after all.

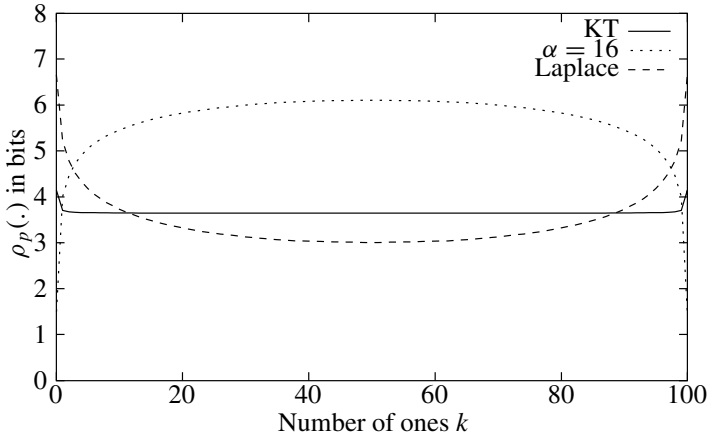


Figure 4.5: The redundancy of the estimator for sequences of length 100.

4.3.2 A MORE GENERAL ESTIMATOR

The Krichevsky-Trofimov estimator performs roughly optimal. The individual redundancy of this estimator on any sequence of length N, compared to its actual probability, is at most $\frac{1}{2}\log_2 N + 1$ bits. This upper bound is independent of the parameter θ of the sequence, but the actual individual redundancy varies with θ. Consider sequences of fixed length N with k ones for $0 \leq k \leq N$. The actual probability of a sequence with k ones, $P_a(x_1^N \text{ containing } k \text{ ones}) = (1-\theta)^{N-k}\theta^k$, is minimized by assuming the optimal parameter, $\hat{\theta} = \frac{k}{N}$, for this sequence. In that case the individual redundancy of the KT-estimator on this sequence reduces to:

$$\rho_p(x_1^N \text{ with } k \text{ ones}) = \begin{cases} \log_2 \frac{1}{P_e(N-k,k)} - \log_2 \left(\frac{N}{N-k}\right)^{N-k} \left(\frac{N}{k}\right)^k, & \text{if } 0 < k < N, \\ \log_2 \frac{1}{P_e(N-k,k)}, & \text{otherwise.} \end{cases} \tag{4.8}$$

The individual redundancy for $\hat{\theta} = \frac{k}{N}$ has been plotted in Figure 4.5 (the solid line) as a function of k for $N = 100$. It achieves its maximum for deterministic sequences (sequences for which the parameter is equal to zero or one). This is an undesirable situation, since a significant number of leaves in the context trees are deterministic, or almost deterministic. Therefore, we introduce a parameter α, with which we can control the redundancy behaviour of the estimator:

$$P_e(X_n = 0 | x_1^{n-1} \text{ has } a \text{ zeros and } b \text{ ones}) = \frac{a + \frac{1}{\alpha}}{a + b + \frac{2}{\alpha}}, \tag{4.9}$$

and similar in case X_n is a one. With $\alpha = 2$ it reduces to the KT-estimator, and with $\alpha = 1$ (see the dashed line) it reduces to the well-known Laplace-estimator. We will use $\alpha = 16$. The resulting estimator has a much lower individual redundancy for deterministic sequences, or for almost deterministic sequences, at the cost of a much higher redundancy for sequences with parameters around 0.5 (see the dotted line in Figure 4.5). Such behaviour is preferable because if a (sub)sequence has a parameter around a half, that sequence cannot be compressed, and

consequently it will be expensive for the CTW algorithm to use such leaves as part of the model. Thus, if possible, the CTW algorithm will prefer to split such nodes further until it reaches more skewed distributions. Experiments in Section 4.5 show that the new estimator indeed achieves a better performance in practice.

Our estimator also uses count-halving. After either the number of zeros or the number of ones reaches a certain fixed value (256 in our case), both counts will be halved and rounded up[3]. This will not only ease the implementation, experiments show that it also improves the performance of the CTW algorithm slightly, because the estimators can now follow the dynamic behaviour of the parameters better.

4.4 IMPLEMENTATION

In principle, each implementation of the CTW algorithm has to perform two actions: searching and computing. First the data structure has to be searched to gather the information from the nodes and leaf on the context path. This action is described in four, closely related, subsections: 4.4.2 discusses how the searching technique handles the fixed amount of memory, 4.4.3 describes the searching technique and the set-up of the data structure, 4.4.4 explains the actual implementation, and finally 4.4.5 evaluates its performance. The second action is the computation of the probabilities. This topic will be discussed first.

4.4.1 COMPUTATIONS

REPRESENTATION

The CTW algorithm computes probabilities. These probabilities are floating point numbers. But the results of floating point operations can differ slightly between different platforms. Since in the CTW decoder the result of an operation should be exactly the same as in the encoder, the CTW algorithm cannot use these floating point operations. We investigated two solutions. These solutions should give a replacement for all operations used in the CTW algorithm: multiplying, dividing and adding probabilities, and computing the (conditional) estimated probability.

First, a floating point number p can be represented by two integers, the mantissa m and the exponent e. The mantissa is an f bit integer of which the most significant bit is a one. The floating point number is now formed by the mantissa with a decimal point placed in front of it, multiplied by 2^e, thus $p = (m/2^f) \cdot 2^e$. The length of the mantissa determines the accuracy of these floating point numbers, and the length of the exponent determines the range of the numbers. For all necessary operations special subroutines have been written which use only integer operations internally.

Secondly, the logarithm of a floating point number with a fixed number of bits, say f, after the decimal point can be stored in an integer [64, 65]. Then a floating point number p is stored as $\lfloor 2^f \log_2 p \rfloor$. Now, the multiplication and division of two floating point numbers reduces to the addition and subtraction of two integers. Only for the addition of two floating point numbers

[3]The counts are rounded up, because then non-deterministic counts will remain non-deterministic.

some additional computations are necessary. Suppose two floating point numbers p and q, with $p \geq q$, need to be added, and only their fixed-point logarithmic representation is available. Then for $2^f \log_2(p+q)$ we find,

$$\begin{aligned} 2^f \log_2(p+q) &= 2^f \log_2 p + 2^f \log_2 \left(1 + \frac{q}{p}\right) \\ &= 2^f \log_2 p + 2^f \log_2(1 + 2^{(2^f \log_2 q - 2^f \log_2 p)/2^f}), \end{aligned} \tag{4.10}$$

with $2^f \log_2 q - 2^f \log_2 p \leq 0$. Integer $i = \lfloor 2^f \log_2 q \rfloor - \lfloor 2^f \log_2 p \rfloor$ can be computed directly with our representation, thus we can compute (4.10) easily by means of a table that contains the integers $\lfloor 2^f \log_2(1 + 2^{i/2^f}) \rfloor$ for all meaningful values of integer i.

Both ways to represent the floating point numbers have been implemented. For both implementations the processor time used for the probability computations has been measured on several test files. These experiments showed that the fixed-point log-implementation is about five times as fast as the implementation with two integers. Therefore we chose to use the fixed-point log-implementation in the final design, although it might complicate the interfacing with other components.

THE ORDER OF COMPUTATIONS

The CTW algorithm cannot be implemented as it is described in Section 4.2.5. The probabilities are rounded or truncated by each operation. The only comprehensive solution to all accuracy problems is if the encoder and the decoder compute the probabilities for both values of the next bit. This guarantees decodability of the code words. Furthermore, if these probabilities are normalized in every node, then small rounding errors will not propagate. Our final design (it resembles [65]) implements these two measures, while it also reduces the computational complexity and the memory complexity, compared to the basic implementation. In every node s of the context tree, our final design uses the following steps to compute the conditional weighted probability (note that all operations are performed in the fixed-point log-domain):

1. Compute, with the β^s in this node s,

$$\beta^{s'}(x_1^n) = \frac{\beta^s(x_1^n)}{a_s + b_s + 2/\alpha},$$

and the terms:

$$\begin{aligned} \left[\beta^s(x_1^n) P_e^s(0|x_1^n)\right] &= \beta^{s'}(x_1^n)(a_s + 1/\alpha), \\ \left[\beta^s(x_1^n) P_e^s(1|x_1^n)\right] &= \beta^{s'}(x_1^n)(b_s + 1/\alpha). \end{aligned}$$

Both the numerator term $\log_2(i + \frac{1}{\alpha})$, and the denominator term $\log_2(i + \frac{2}{\alpha})$ of the conditional estimated probability have been stored in tables, for all possible values of integer i.

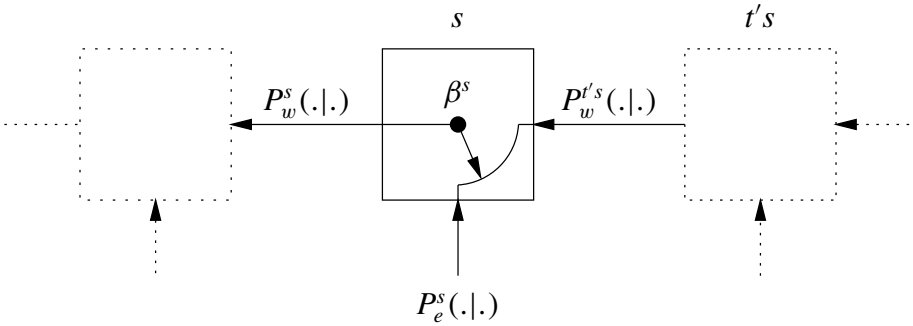


Figure 4.6: The β^s-factor operates like a switch.

2. Compute an intermediate result η^s from the previous results:

$$\eta^s(x_1^n, 0) = \left[\beta^s(x_1^n) \cdot P_e^s(0|x_1^n)\right] + P_w^{t's}(0|x_1^n), \tag{4.11}$$
$$\eta^s(x_1^n, 1) = \left[\beta^s(x_1^n) \cdot P_e^s(1|x_1^n)\right] + P_w^{t's}(1|x_1^n), \tag{4.12}$$

   in which $t' \in \mathcal{X}$ and $t's$ denotes the child of s on the context path.

3. Now, compute the conditional weighted probabilities (and normalize at the same time):

$$P_w^s(0|x_1^n) = \frac{\eta^s(x_1^n, 0)}{\eta^s(x_1^n, 0) + \eta^s(x_1^n, 1)},$$
$$P_w^s(1|x_1^n) = \frac{\eta^s(x_1^n, 1)}{\eta^s(x_1^n, 0) + \eta^s(x_1^n, 1)}.$$

4. Update β and store the new value in this node:

$$\beta^s(x_1^{n+1}) = \frac{\left[\beta^s(x_1^n)\, P_e^s(x_{n+1}|x_1^n)\right]}{P_w^{t's}(x_{n+1}|x_1^n)}. \tag{4.13}$$

5. Increment a_s if $x_{n+1} = 0$, or increment b_s if $x_{n+1} = 1$. If the incremented count reaches a fixed maximum (256 in our implementation), both counts are halved and rounded up.

How does this new scheme work? The key element is the new factor $\beta^s(.)$. From a more abstract point of view one can imagine that the nodes on the context path from the leaf to the root node form a chain. Each node s in this chain has as its inputs two estimates of the probability distribution of the new bit: the conditional weighted probability $P_w^{t's}(.|.)$ from its child node $t's$ on the context path, and the conditional estimated probability $P_e^s(.|.)$ computed in this node. The factor β^s acts like a switch that determines how these two conditional probabilities will be mixed (see Figure 4.6). This is realized by equations (4.11) and (4.12). The state of the switch is determined by the performance of these two estimates in the past (equation 4.13).

The factor β is initialized to 1 in new nodes. At any point during the coding of the source sequence, the β^s in a node s represents,

$$\beta^s(x_1^n) = \frac{P_e^s(x_1^n)}{\prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n)}. \tag{4.14}$$

Thus for the intermediate result $\eta^s(x_1^n, 0)$ we find:

$$\begin{aligned}
\eta^s(x_1^n, 0) &= \frac{P_e^s(x_1^n)}{\prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n)} \cdot P_e^s(0|x_1^n) + P_w^{t's}(0|x_1^n) \\
&= \frac{2}{\prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n)} \left(\frac{1}{2} P_e^s(x_1^n) P_e^s(0|x_1^n) + \frac{1}{2} \prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n) P_w^{t's}(0|x_1^n) \right) \\
&= \frac{2}{\prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n)} \left(\frac{1}{2} P_e^s(x_1^n, 0) + \frac{1}{2} \prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n, 0) \right),
\end{aligned} \tag{4.15}$$

and similar for $\eta^s(x_1^n, 1)$. Therefore, the intermediate result is just the weighted probability, multiplied by a factor which is independent of the new symbol x_{n+1}. By computing the conditional weighted probabilities (step 3), these intermediate results are normalized, and the constant factor is eliminated.

Introducing the β-factor has some advantages. First, it clearly shows the workings of the CTW algorithm: in every node the past performance directly controls the weights assigned to the estimate of that node and the estimate of the child nodes. Secondly, the memory usage per node s of the context tree is reduced because only the factor β^s has to be stored (instead of two probabilities). And finally, the computational work is reduced. The algorithm described above is very efficient. It needs only three table-accesses, three additions, two multiplications and four divisions to compute the new conditional weighted probability in each node.

The decoder doesn't know the new symbol when it computes the weighted probability. It will (just like the encoder) compute both the probability in case the symbol is a zero and a one, and it will assume that the symbol is a zero, and update the context tree accordingly. But at the same time it will store the values in case it was a one in a "back-up" array (together with the address of the node), so that if the new symbol is indeed a one, it can easily overwrite the incorrectly updated records with the correct versions.

ACCURACY

First, the number of bits f after the decimal point in the fixed-point log implementation has to be determined. For the memoryless estimators in the nodes and the leaves we will demand that the accuracy f is sufficiently high such that for each possible set of counts a_s and b_s in a record s, $P_e(0|a_s + 1, b_s) \neq P_e(0|a_s, b_s + 1)$ holds. If, due to a lack of accuracy, these probabilities are the same for some a_s and b_s, then the next symbol will not influence the estimated probability distribution in this node or leaf, regardless of its value. Experiments show that with count halving after 256 counts, and with $\alpha = 16$, at least $f = 8$ fractional bits are required. Since the conditional weighted probability in the root node $P_w^{\lambda}(x_{n+1}|x_1^n)$ is just a linear combination of the

estimated probabilities along the context path, it seems reasonable to use the same accuracy for the computation of the weighted probabilities too.

Next, the sizes of the various tables should be set. Since in the final design the counts are halved if one of them reaches 256, the numerator table (containing integers $\lfloor 2^f \log_2(i + \frac{1}{\alpha})\rfloor$) needs 256 entries and the the denominator table (with integers $\lfloor 2^f \log_2(i + \frac{2}{\alpha})\rfloor$) needs 511 entries. The size of the table used for the addition of two floating point numbers (see equation (4.10)) also has to be determined. For all i for which $2^f \log_2(1 + 2^{i/2^f}) < 1$ (we truncate the numbers) this table will contain a 0 and does not need to be stored. For $f = 8$, the table requires 2183 entries (from 0 to -2182).

Finally, the fraction β^s has to be stored in every node. It is represented with 8 fractional bits, but how many bits in front of the decimal point are necessary? The factor β^s acts like a switch that determines how the conditional probabilities $P_w^{t's}(.|.)$ and $P_e^s(.|.)$ will be mixed. These two probabilities are mixed in step 2 of the algorithm (see equations (4.11) and (4.12)):

$$\eta^s(x_1^n, 0) = \left[\beta^s(x_1^n) \cdot P_e^s(0|x_1^n)\right] + P_w^{t's}(0|x_1^n), \tag{4.16}$$

for $x_{n+1} = 0$, and similar for $x_{n+1} = 1$. The range of β^s should be such that it can completely choose one of these two conditional probabilities without interference from the other one, independently of the values of these two probabilities. The most skewed estimated probability distribution $P_e^s(.|.)$ is obtained if one source symbol is observed 255 times, and the other symbol did not occur yet. In that case, with $\alpha = 16$, the most probable symbol has probability $4081/4082$, while the other symbol has probability $1/4082$. This is also the most skewed probability distribution of the conditional weighted probability $P_w^{t's}(.|.)$ of the child node, because that probability is a linear combination of the estimated probabilities of the longer contexts. One extreme situation occurs if the switch is completely turned to estimate $P_e^s(.|.)$, while this estimate is $P_e^s(0|x_1^n) = 1/4082$ and $P_w^{t's}(0|x_1^n) = 4081/4082$. From the previous discussion we find that the term $P_w^{t's}(0|x_1^n)$ is negligible if

$$2^f \log_2\left(\beta^s(x_1^n)\frac{1}{4082}\right) - 2^f \log_2\left(\frac{4081}{4082}\right) \geq 2183,$$

for $f = 8$. Consequently, β^s should be at least 1.5 million, or more precisely $2^f \log_2 \beta^s$ should be about 5254. The other extreme situation, is exactly the reverse from this one, and as a result the range of the integer $\lfloor 2^f \log_2 \beta^s \rfloor$ is about $[-5254, 5254]$. Therefore, storing β in its fixed-point log representation requires one sign bit, five integer bits, and eight fractional bits, thus 14 bits in total, or, for sake of convenience, one word of two bytes.

The two extreme situations described above, will almost never happen in practice. Experiments will show that a smaller range for β will result in a better performance.

4.4.2 SEARCHING: DEALING WITH A FIXED MEMORY

One full context tree for D 256-ary context symbols has 256^D leaves. Thus it seems unfeasible to store 255 different full context trees of depth three or more. On the other hand, consider context trees of depth 8. There are $1.84 \cdot 10^{19}$ different combinations of 8 bytes, thus in a text file of a few

hundred kilobytes only a tiny fraction of all possible combinations will actually occur. Therefore the context trees will be stored in a dynamical structure, which allows us to control the growth of the context trees. We will use three techniques.

DYNAMICAL GROWTH OF THE CONTEXT TREE

First, the context tree can be grown dynamically. In a node s, the CTW algorithm uses only information obtained from the node itself and the conditional weighted probability $P_w^{t's}(.|.)$ as computed in the child $t's$ on the context path. All possible other children ts (with $t \neq t'$) of this node s can be ignored for the computations in s. As a result, only the nodes and leaves s that occur in the source sequence have to be created.

UNIQUE PATH PRUNING

Secondly, if a context becomes unique then its descendants do not have to be created. A context s becomes unique if all contexts that occurred so far in the source sequence and that had s as a suffix, are completely identical. In that case only one path occurred from node s to a leaf at maximum depth D so far. In all nodes s on this unique path we find:

$$P_w^s(x_1^n) = P_e^s(x_1^n),$$

for all $n = 1, 2, \ldots, N$. Thus, in these nodes $\beta^s(x_1^n) = 1$ and consequently $P_w^s(x_n|x_1^{n-1}) = P_e^s(x_n|x_1^{n-1})$. As a result, only the first node s of a unique path has to be created, and in this node s only the probability $P_e^s(.)$ has to be computed. A small problem arises if a node s has been unique so far, and the context of the new symbol has s as a suffix too, but has a different continuation. Then, the context path through node s has to be extended until the two contexts differ. But the contexts can only be compared, and possibly extended, if the tail of the unique context s is stored somewhere. This is investigated in Section 4.4.3.

PRUNING VERSUS FREEZING

Finally, a pruning technique has to be applied. Although the two previous techniques significantly reduce the number of nodes and leaves that will be created, the total number of nodes and leaves will still increase as more of the source sequence is processed. But the final design may use only a limited amount of memory. Thus at some point the data compression program uses all available memory, and it cannot create any new nodes or leaves. There are two solutions to this problem.

First, the compression program can freeze the context tree at this moment. If a node s needs to be extended, but that is impossible due to a lack of memory, then one can just use that node as a leaf instead. This simple and fast solution has the disadvantage that it cannot adapt to a new model anymore, if the statistics in the source sequence change.

Secondly, a real pruning technique can be applied. Here, if a new node or leaf needs to be created and the context trees already use all memory, then a pruning algorithm selects one node from the context trees and removes this node, together with all its descendants. The pruned node

is then changed into a leaf. If necessary this leaf can be extended again later, but the counts collected in this leaf so far will then be missing from its subtree. We experimented with different set-ups of such a pruning algorithm. It had several parameters:

- Which node will be pruned? The subtree in the node s with the highest quotient $\beta^s(.) = P_e^s(.)/\prod_t P_w^{ts}(.)$ contributes least to the total weighted probability. Alternatively, one might compute this quotient per node and leaf in the subtree. This reflects the contribution of the subtree from node s, per used node.

- If the available memory is tight, all nodes in use might actually be part of the real model. Newly created nodes will start with an (almost) neutral quotient, and will be pruned almost immediately, which prevents the model from changing adaptively. This is solved by protecting a fixed percentage of the newly created nodes from pruning. Alternatively, one might consider protecting a fixed percentage of the newly created *and* recently updated nodes. This would protect also the frequently visited parts of the trees.

The major difficulty for the implementation is that the pruning program needs two different data structures over the nodes of the context tree. One structure orders the nodes and leaves in context trees, and the other structure should allow the pruning program to identify the "worst" node efficiently. These two structures can be implemented independently, but this may result in additional references between these two structures, therefore we decided to integrate the two structures. This puts some constraints on the data structures, because the two structures now share the same records, and one structure cannot move a record around or remove a record, without destroying the coherence in the other structure. We implemented pruning in a development version of our program. That version uses pointers, which allowed us to solve this problem easily, by modifying the data structures only through changing the pointer references: the nodes themselves did not move through the memory. Although the records are shared by the two structures, the size of the records still increases. As will be shown later, our final design CTW algorithm needs records of 6 bytes each. If a fast independent structure has to be placed on top of these records, the size of the records will probably double (an efficient search structure for the "worst" node, e.g. a heap, requires two or three additional pointers per node). If also the number of descendants has to be counted, then the record size will increase even more.

Various set-ups have been simulated. The pruning algorithms that prune the node with the highest β and with the highest β per descending node or leaf gave a comparable performance. Since the latter technique uses more memory per record and it needs more computations per pruning operation, we will use β as criterion. Next, we chose to protect only new nodes because it is most of the time marginally (less than 1 %) better than protecting both new and updated nodes, and in our case the simulation program is faster too. Finally, we set the percentage of protected nodes at 20 %. Less than 10 % resulted in poorer performance, and higher percentages might force the algorithm to start pruning good nodes, since all the bad nodes are still protected. This is of course not the optimal set-up for every situation, but it seems a good general set-up.

The CTW algorithm has been simulated with both the freezing scheme as well as the pruning scheme. Tables 4.1 and 4.2 summarize the estimated performance of these schemes on the English text file book2 (610856 bytes) from the Calgary corpus, and a text file (1591161 bytes)

Table 4.1: The results of a pruning algorithm and the freeze algorithm on file book2.

	maximum number of records							
	100,000		250,000		500,000		1,000,000	
	bits/sym	D^*	bits/sym	D^*	bits/sym	D^*	bits/sym	D^*
freeze	2.328	3	2.086	4	1.977	4	1.893	5
prune	2.027	6	1.939	6	1.889	6	1.853	8

Table 4.2: The results of a pruning algorithm and the freeze algorithm on the Jane Austen books.

	maximum number of records							
	250,000		500,000		1,000,000		2,500,000	
	bits/sym	D^*	bits/sym	D^*	bits/sym	D^*	bits/sym	D^*
freeze	1.938	4	1.865	5	1.802	5	1.763	6
prune	1.836	5	1.795	6	1.767	7	1.743	9

containing two books, "Emma" and "Pride and Prejudice", by Jane Austen, respectively. A CTW simulation with depth $D = 8$ and exactly the same set-up, but with unbounded memory, compresses book2 to 1.835 bits/symbol with 2,523,412 records and the two books by Jane Austen to 1.741 bits/symbol with 5055696 records. The tables show for each given number of records the performance of the simulations ("freeze" freezes the trees and "prune" prunes the trees) in bits/symbol, together with the optimal maximum depth of the context trees D^* that achieves this performance. It is interesting to note, that choosing a good depth is crucial for the "freeze" algorithm, but that the maximum depth of the tree in the "prune" algorithm does not influence the results much. At first glance, pruning seems to perform much better than freezing. But, to make a fair comparison, the pruning algorithm should be compared with a freezing algorithm with twice the number of records, to compensate for the increased record size. For both texts the pruning algorithm with 250,000 records performs less than 2 % better than the freezing algorithm with 500,000 records, and the pruning algorithm with 500,000 records performs comparable to the freezing algorithm with 1,000,000 records. It appears that there exists a "core model" of about 500,000 records for English texts. Once this core model can be stored, only minor improvements in the rate can be obtained by increasing the number of records further. For the pruning algorithm 500,000 records are sufficient, while the freeze algorithm requires more records before it has captured the core model. But the difference in number of necessary records is cancelled out by the difference in size of the records.

The second series of simulations uses the file obj2 (246814 bytes) from the Calgary corpus. This is a binary executable. While the statistics in a text file are more or less stationary, here the statistics change drastically between different parts of the file. The simulation program with unbounded memory and with depth 8 compresses this file to 2.391 bits/symbol and it requires 1,731,277 records. Table 4.3 summarizes the results of the freezing algorithm and the pruning

Table 4.3: The results of a pruning algorithm and the freezing algorithm on file obj2.

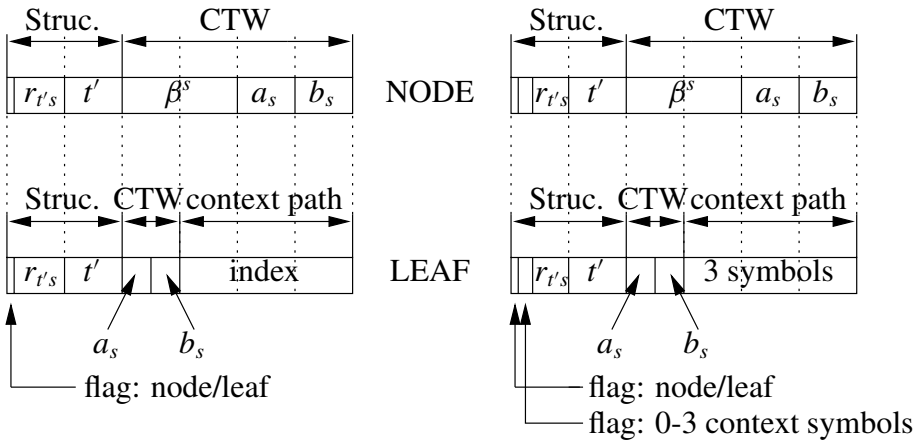
	maximum number of records									
	80,000		170,000		340,000		680,000		1,360,000	
	bits/sym	D^*	bits/sym	D^*	bits/sym	D^*	bits/sym	D^*	bits/sym	D^*
freeze	3.772	2	3.180	2	2.860	3	2.549	4	2.433	6
prune	2.836	4	2.530	8	2.450	8	2.409	8	2.381	12

algorithm. Now, the pruning algorithm with 340,000 records still performs 4 % better than the freezing algorithm with 680,000 records (about 40 % of the total number of records for $D = 8$). For a good performance, the CTW algorithm should be able to create a different "core" model for the different parts in this file. But the freezing algorithm is too greedy and cannot adapt anymore towards the end of the file, even if the model changes significantly, while the pruning algorithm can reclaim old records and can continue to adapt. Thus, while for text files the pruning algorithm is only effective for small numbers of records, for this kind of files, it is effective for a much larger range of records.

We decided to simply freeze the tree in our final design, once the memory is filled. The pruning algorithm is in general more computational intensive than the freezing algorithm, and it is more complex to implement. The main objective of the pruning algorithm was to deal with the fixed amount of memory. From the simulations it is clear that the pruning algorithm is only significantly better if the amount of memory is very low and the freezing algorithm cannot create the entire "core" model. Since the final design can accommodate 5.5 million records, a freezing algorithm with a good maximum depth will perform comparable. The only additional advantage of the pruning algorithm is its robustness for the setting of the maximum depth, while this setting is crucial for the performance of the freezing algorithm. In this respect, the pruning algorithm could have an alternative use. It grows the models selectively, thus it can use much deeper context trees than the freezing algorithm, with the same memory complexity. Tables 4.1, 4.2 and 4.3 show indeed that the optimal depth of the pruning algorithm is significantly higher than the optimal depth of a comparable freezing algorithm. But, from these tables it is also clear that even if a lot of records are available, the pruning algorithm does not succeed in achieving a significant gain in performance from its longer contexts.

4.4.3 SEARCHING: SEARCH TECHNIQUE AND DATA STRUCTURES

A record in a context tree consists of two or three parts: a CTW part for CTW specific information, a structural part for information necessary for maintaining the data structure, and in leaves a third part for reconstructing unique context paths. These three parts will be discussed independently in the following subsections. In our final design we will store all records in six bytes (see Figure 4.7). Both nodes and leaves have two bytes structural information. In nodes the remaining four bytes are used for CTW information, while in leaves only one byte is used for CTW information and three bytes for context information.



(a) Record structure with indexes (b) Record structure with symbols

Figure 4.7: The different record structures as used in the final design.

THE CTW INFORMATION

In a node s, the CTW algorithm stores counters a_s and b_s and the fraction β^s. Because our count halving technique limits the counts to 255, they can be stored in one byte each. For the β-factor in the nodes we already found that a range of $[-5254, 5254]$ is sufficient, so we can store β in two bytes. Therefore, the CTW information in nodes fits in four bytes in total.

In a leaf s, the CTW algorithm stores only the two counters a_s and b_s. Simulations show that the vast majority of the leaves has only a few counts. After compressing each file in the Calgary corpus, the counts in the leaves were examined. In practice the leaves in which at least one of the symbols occurred 16 times or more, accounted for less than 1.5 % of all counts in the leaves. Therefore, each of the counts in a leaf can be carelessly stored in four bits, thus in one byte for both. Alternatively, it is possible to construct a state machine with less than 256 states, that can count all combinations of a_s and b_s for up to and including 21 symbols. This could result in a superior performance, because the simulations show the leaves with in total 22 or more counts, account for less than 0.2 % of all counts in the leaves. Both techniques have been implemented and will be compared later. Regardless of which solution will be chosen, the CTW information in leaves requires one byte.

UNIQUE PATH RECONSTRUCTION INFORMATION IN LEAVES

If the CTW algorithm finds a leaf that is not at the maximum depth, then it should be at the beginning of a unique path. To compare the remaining part of the context with the new context, and to allow extension of the context tree from this leaf on, the leaf needs to contain additional information concerning the following context symbols. We investigated two different techniques to store these additional context symbols. Because we require that all types of records have the same size (to ease the design of the search technique), the unique context path information in each node should not use more than three bytes.

1. The entire source sequence can be stored in a buffer, and each leaf can simply refer to the position in the file at which this context occurred last. This has the advantage that the context paths can be reconstructed completely, but it has the disadvantage that an additional buffer is required. A compromise would be a buffer of limited length, in which only the most recent part of the source sequence is stored. In this implementation the three bytes are used as an index.

2. Alternatively, the first few still unused context symbols can be stored in the leaf itself. The advantage of this technique is that an additional buffer is not necessary, but the disadvantage is that the limited number of available context symbols causes some problems. Suppose the following three context symbols can be stored in a record. First, if the context of the new symbol differs from the context stored in this leaf, the context path will be extended until each context becomes unique again. Note that the new leaf that corresponds to the old unique context, has now fewer than three context symbols. Secondly, if the context of the new symbol matches the few context symbols in the leaf we, possibly wrongly, assume that the entire context is identical and if possible we even replenish the number of context symbols stored in this leaf to its maximum. This has the disadvantage that possibly counts of different contexts are collected in one leaf. These spoiled counters may influence the performance negatively. Still we chose this implementation since it is fast, and it keeps the number of records low.

   Two possibilities seem useful: at most two context symbols can be stored (which would allow more accurate counters), or at most three context symbols can be stored. Simulations on the Calgary corpus show that the loss from the incomplete context path reconstructions is about 0.014 bits/symbol if two context symbols are stored and 0.004 bits/symbol if three context symbols are stored. We chose to use the implementation with three context symbols.

There is a clear trade-off between the two techniques. The technique with a limited buffer can reconstruct the context paths by any number of symbols, but for a limited period of time only, while the alternative technique can reconstruct the context paths only by a limited number of context symbols, but these are stored for an unlimited time. These two techniques are compared in Section 4.5.

Figure 4.7 shows the differences in record structure between these two choices. Note, that regardless of which technique will be chosen, the CTW algorithm still has to be able to distinguish between nodes and leaves. This costs one extra bit. Furthermore, the second technique needs to register how many context symbols are stored in a leaf (from 0 to 3). This costs two additional bits. Both flags are included in the figure.

THE STRUCTURAL INFORMATION AND THE SEARCHING TECHNIQUE

The structural information is used to find a specific node or leaf in the context trees. Previous implementations ([48]) of the CTW algorithm often use a binary search tree to find the desired child of a node. These implementations require three pointers per record of which the vast

majority will not be used. This causes an enormous overhead. In [64] an implementation with a hash table is presented which reduces the overhead enormously. For our design we will use an even more efficient implementation (due to Tjalkens [45, 58]).

The final design uses a hashing technique based on open addressing. There is one hash table of fixed size that contains all records. The efficient hashing technique relies on the incremental search behaviour of the CTW algorithm. As the CTW algorithm traverses along a context path in a context tree, it always searches for the record that corresponds to the context that is one symbol longer than the previous context. This behaviour is used in the following way. First the root node of every context tree is created in the hash table and the index of each root node is stored in a table. The CTW algorithm knows from the decomposition tree which context tree to use next, and from this table it knows the index of the corresponding root node. Furthermore a second table, the offset table, is created with for each possible context symbol a different offset (step size). Now, suppose the CTW algorithm already found the record corresponding to context s in a certain context tree and it is looking for the child node representing context $t's$. This child node is created at an index which is at a multiple of the offset of t' from record s. First the index which is at one times the offset (modulo the hash table length) from s is probed, and if this index is used by a record corresponding to a different context, then the index at twice the offset is probed. This process continues until the correct record has been found, until an empty slot has been found (the child does not exist yet), or until 31 indexes (the maximum number of probes) have been probed unsuccessfully. In the last case this record is lost.

The structural information in a record should be such that the hashing technique can determine whether the record it just probed is indeed the desired child of a node, or not. If a (child) record corresponding to a context $t's$, contains its most recently used context symbol t' (which specifies the offset) and the number of steps $r_{t's}$ that are necessary to reach this record from the record corresponding to its parent node s, then each record uniquely specifies the address of its parent node. Thus as soon as one of the probed indexes has the correct number of steps and the correct context symbol t', it has to be the desired child node. The structural information requires one byte (the next context symbol) and five bits (for the number of steps).

Figure 4.7 shows the entire structure of the records. The records fit completely in six bytes.

4.4.4 SEARCHING: IMPLEMENTATION

The performance of the hashing technique is judged by two important properties: the average number of probes needed to find a record, and the load factor, which is the percentage of records used. In our design the behaviour of the hashing technique is controlled by only two functions. First, the function that computes the offset table which should contain a different offset (step-size) for each context symbol. Secondly, the function that computes the 255 different starting indexes for the root nodes. In order to find a record with a low average number of probes, combined with the ability to achieve a high load level, the indexes of the root nodes and the offsets should be randomized. These randomization functions in our final design are slightly modified versions of those in [64].

The final design uses a memory block of 2^B bytes to store the hash table. Because the records use 6 bytes the last 2 or 4 bytes of the memory block will not be used. The function that computes

$Pperm$[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 4.4: Pearson's permutation table.

the "random" address A_T of a root node of context tree number T is described below.

Compute address root node:

- Compute a 32 bits random number R_{32}, based on T, the number of this context tree (we follow the C-language convention and use the symbol "&" to denote a logical bit-wise and, and the symbol "$\ll$" to denote a logical shift left):

$$R_{32}(T) = ((Pperm[(T+3)\&255]) \ll 24) + ((Pperm[(T+2)\&255]) \ll 16) + \\ + ((Pperm[(T+1)\&255]) \ll 8) + Pperm[T].$$

- Compute a random B bits multiple of 8:

$$R_B(T) = (R_{32}(T)\&(2^{B-3} - 1)) \ll 3.$$

- Address A_T is now, $A_T = \lfloor R_B(T)/6 \rfloor * 6$.

First it generates a 32 bits random number, R_{32}, by taking 4 consecutive bytes from Pearson's permutation table (see Table 4.4). The $B-3$ least significant bits of this number are shifted three bits to the left to obtain R_B, a random B bits multiple of eight, smaller than 2^B. Now the largest multiple of 6, smaller than this value is used as next root node. This routine guarantees that for $B \geq 11$ the 255 root nodes get a different address.

The offsets are computed in a similar fashion, but the outcomes of this randomization function should be checked more carefully. At least, the offsets should not be periodic in 31 steps, and it might also be preferable if every number of steps and every offset gives an unique total offset. In that case the children of one parent node will never collide with each other. The offset $O(t)$ of symbol t is computed as:

Compute offset:

- Compute a 32 bits random number, based on the value of the context symbol t:

$$R_{32}(t) = ((Pperm[(t+3)\&255]) \ll 24) + ((Pperm[(t+2)\&255]) \ll 16) + \\ + ((Pperm[(t+1)\&255]) \ll 8) + Pperm[t].$$

- Compute the offset $O(t) = (R_{32}(t)\&(2^{B-3-ES}-1)) * 6$.

In this function ES is an additional variable, Extra Shifts. Without this variable, the $B-3$ least significant bits of R_{32} are taken and multiplied by six. This would set the largest offset to almost 0.75 times the size of the hash table. Experiments show that keeping the offsets smaller, improves the search speed slightly (less than 1 %), and achieves a slightly higher load factor once the hash table is almost completely filled. We use $ES = 2$, which reduces the largest offset to less than 0.1875 times the hash table size. As a result, B should now be at least 14 (9 bits to get a different offset for each symbol, keeping in mind that we cannot use offset 0, plus 3 bits to divide by eight, plus 2 bits Extra Shift)[4]. Experiments show that with this function, and for all relevant sizes of the hash table (from $B = 14$ up to $B = 26$) the offsets are aperiodical. Unfortunately, even for large hash tables ($B = 24$ and greater), not every combination of a number of steps and an offset will result in a unique address. Thus two children of one parent node can still collide. But since this occurs very rarely and only for very few combinations (10 and 9 combinations for $B = 24$ and $B = 25$, respectively), it will not influence the hashing performance much.

If a child t' of parent node s at address A has to be found, the hashing technique performs the following steps. They form a straightforward implementation of the collision resolution technique described before.

Find child node:

- Parent node s has address A.
- For step $r_{t's} = 1, 2, \ldots, 31$ do:

[4]Interestingly, even with no extra shifts (ES=0) and with ES=1, the minimum hash table size is still 14 bits, because for smaller hash table sizes some of the offsets will be periodic.

- ⋆ Compute new address: $A = A + O(t') \bmod \lfloor 2^B/6 \rfloor * 6$.
- ⋆ If A is empty then create new leaf here with as keys the symbol t' and the number of steps $r_{t's}$.
  Return A and exit loop.
- ⋆ If the keys of the record at A match the current context symbol t' and the current number of steps $r_{t's}$, then this is the correct child node.
  Return address A and exit loop.

- If no address A found, then record is lost.

4.4.5 SEARCHING: PERFORMANCE

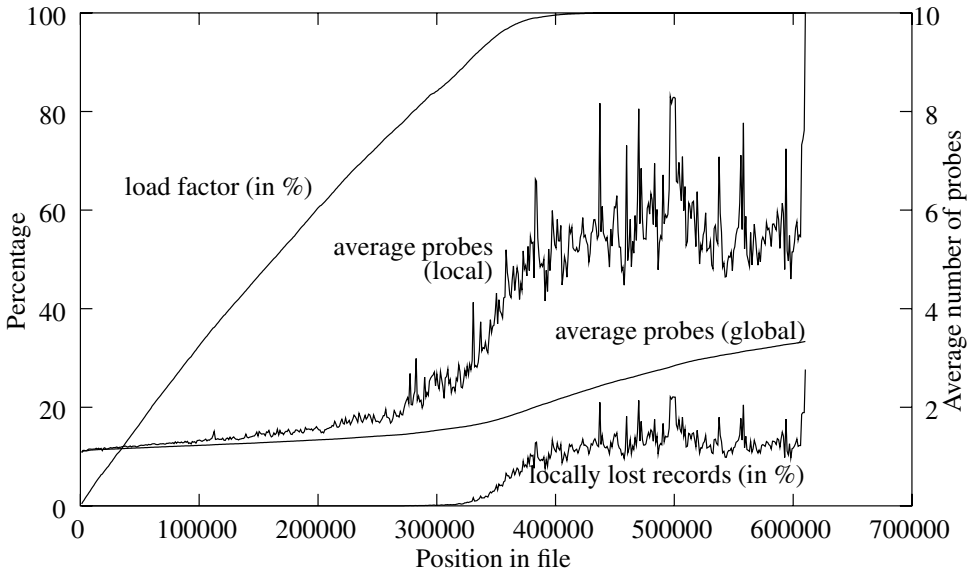
To evaluate the speed of the hashing technique we measured its performance on several files of the Calgary corpus (see Section 4.5.1). The behaviour on these files was very similar, so we will only show the results for book2, an English text file of 610856 bytes. The size of the hash table and the maximum depth have been chosen such that about halfway through the file, the entire hash table will be filled. The experiments have been done both with the technique that stores the context symbols in a leaf, and with the technique that uses a buffer. The results are plotted in Figure 4.8. Both plots have four curves:

- the load factor: the percentage of the hash table that has been used,
- average probes (global): the average number of probes per search so far,
- average probes (local): the average number of probes per search over the last 1024 symbols, and
- locally lost records: the percentage of lost records per search over the last 1024 symbols.

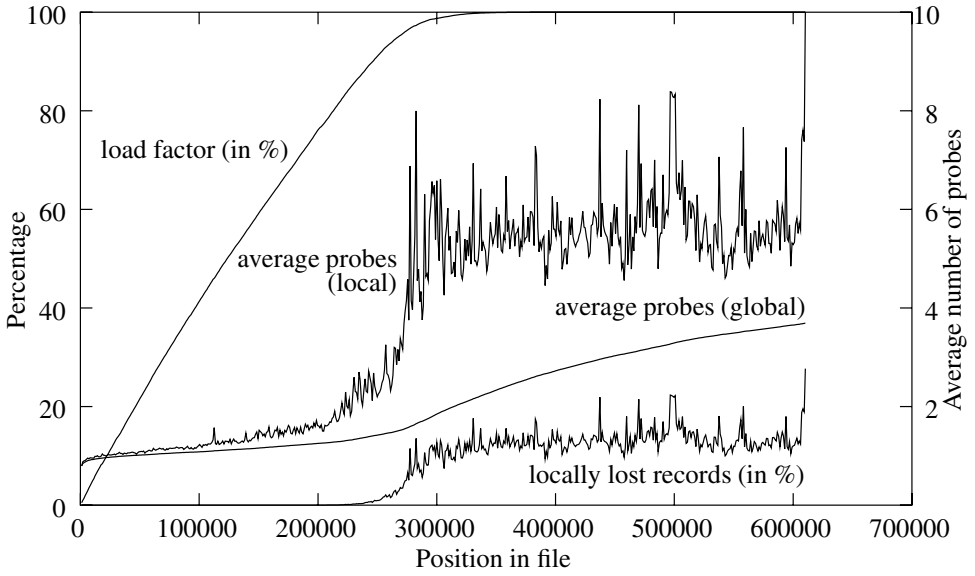
The percentage is plotted on the primary y-axis on the left, the number of steps on the auxiliary y-axis on the right.

The plots clearly show that storing at most three context symbols in the leaves restrains the growth of the context trees. With an index in each leaf, the hash table is filled much faster. Furthermore, the search routine needs on average only slightly more than one probe per search as long as the hash table is filled for less than 80 %. After that, the average number of probes per search quickly increases to about 6 probes per search operation. From a load factor of about 90 % on, the percentage of lost records starts to increase to about 20 %. This demonstrates that for a load factor of up to 80 % the hashing technique works very efficient, and for a load factor of up to 90 %, it still works almost flawless (negligible number of lost records). Furthermore, the performance of this implementation is almost identical to the one described in [64].

The average search speed drops significantly when the hash table is almost completely filled. There are several possible techniques to prevent this. The most straightforward technique would be to freeze the tree once the hash table is filled for, say 95 %. From the plots in Figure 4.8 it is



(a) With context symbols in the records.



(b) With an index in the records.

Figure 4.8: The performance of the hashing technique on book2.

clear that the load factor increases more or less linearly. The slope of the load factor is determined by the maximum depth D of the context trees. Experiments show that adjusting the maximum depth such that the load factor reaches its maximum near the end of the source sequence results in the best performance, combined with a high search speed. Such an adaptive maximum depth of the context trees has not been implemented.

4.5 FIRST EVALUATION

4.5.1 THE CALGARY CORPUS

Throughout this thesis we will use the Calgary corpus [7] as our set of test files. The Calgary corpus consists of the 14 files listed in Table 4.5. The total size of this corpus is 3141622 bytes.

The CTW algorithm has many parameters and it is difficult to set all parameters such that the exact global maximum is achieved. To show that our set-up at least achieves a local optimum, we use a reference set-up, from which we will vary each parameter individually, while keeping the other ones the same. Our reference set-up uses the WNC arithmetic coder with 16 bits accuracy. It uses a CTW algorithm with the ASCII decomposition and context trees of depth $D = 8$. The hash table has a size of 32 MBytes. We will also use a file buffer of 1 MBytes, which is sufficient to store each file of the Calgary corpus completely. The remaining parameters are the ones that will be varied one-by-one: the factor β is bounded to the interval $[-2500; 2500]$, the factor α in the estimator is set to 16 and in the leaves of the context trees the counts are stored in 4 bits.

In the remainder of this section we will first fix the estimator (e.g. determine α), then find the best range for β an finally choose between the two solutions for the unique path pruning. Once these choices are made, we will compare our program to the two competitive programs, "state selection" and PPMZ-2.

4.5.2 FIXING THE ESTIMATOR

First we will fix the estimator. Table 4.6(a) shows the results (length of the compressed corpus in bytes) of compressing all 14 files of the Calgary corpus with the CTW algorithm in which α varies between 10 and 22, and in which the counts in the leaves in the context trees are either stored in 4 bits each, or with the state-machine. From this table it is clear that $\alpha = 20$ results in the highest compression rate. But other experiments show that for long text files, $\alpha = 14$ will give a better performance. Since text files are specifically mentioned in the project description, we decided to use $\alpha = 16$ as a compromise. For each value of α storing the counts with a state machine performs slightly better than storing the counts in 4 bits. But the gain is very small (for $\alpha = 16$ about 0.00022 bits/symbol), while it also results in a small increase in complexity. Therefore we decided not to use the state machine in the final design.

Table 4.5: The files in the Calgary corpus.

name	size	type	description
bib	111261	text	725 references of books and papers
book1	768771	text	book by Thomas Hardy, "Far from the Madding Crowd"
book2	610856	text	book by Ian Witten, "Principles of computer speech"
geo	102400	binary	seismic data represented as 32 bits numbers
news	377109	text	USENET news articles
obj1	21504	binary	VAX executable of progp
obj2	246814	binary	Apple Macintosh executable
paper1	53161	text	paper by Witten, Neal and Cleary, "Arithmetic coding for data compression"
paper2	82199	text	paper by Witten "Computer (In)security"
pic	513216	binary	bitmap of a black and white picture
progc	39611	text	C source code
progl	71646	text	Lisp source code
progp	49379	text	Pascal source code
trans	93695	binary	transcript of a terminal session

4.5.3 FIXING β

The next parameter is the range of β. From Section 4.4.3 we know that the range does not have to exceed $[-5250, 5250]$. Table 4.6(b) summarizes the results of the CTW program on the Calgary corpus for several ranges of β. From this table it is clear that the compression rate is highest if the absolute value of β is bounded to 2500.

4.5.4 FIXING THE UNIQUE PATH PRUNING

Now, the internal parameters of the CTW algorithm have been set and an important design decision has to be made. The CTW algorithm can either store up to three context symbols in each leaf to extend the context path later, or it can store an index referring to a limited file buffer. The performance of these two techniques is measured on five files from the Calgary corpus: book2, geo, obj2, paper1 and pic. In total four different settings will be used. First, the reference setting ("1 MBytes") uses a file buffer of 1 MBytes. The second setting ("no") does not use a file buffer, but stores at most three context symbols in every leaf. The third setting ("$<$ 1/3") uses a file buffer that is at most a third of the length of the file. Since the length of the file buffer (in our implementation) has to be a power of two, we use file buffers of 128 kBytes, 32 kBytes, 64 kBytes, 16 kBytes and 128 kBytes for the files book2, geo, obj2, paper1 and pic, respectively. Finally, the fourth setting ("16 kBytes") uses a very small file buffer of only 16 kBytes.

For these experiments we use a hash table of 64 MBytes, such that the hashing technique can create every required record and will not influence the performance comparison. The performance of these four settings is measured for two depths of the context trees: depth $D = 6$

Table 4.6: Fixing the parameters of the CTW algorithm

(a) Fixing the parameters of the estimator.

α	4 bits bytes	state mach. bytes
10	756490	756343
12	755511	755392
14	754860	754757
16	754535	754447
18	754440	754361
20	754369	**754299**
22	754399	754339

(b) Fixing β

β	bytes
2000	755348
2500	**754535**
3000	755486
3500	756487
4000	757276

and depth $D = 10$. The results can be found in Tables 4.7 and 4.8. For depth $D = 6$, the CTW algorithm that does not have a file buffer performs almost as good as the reference setting. This is to be expected, since each leaf can contain up to three context symbols, and the maximum depth of the context trees is only six. Therefore, only few contexts will get lost. The two set-ups with a limited file buffer perform slightly worse. For depth $D = 10$, the situation is different. Now, the CTW algorithm with only a limited file buffer ($< 1/3$) performs best compared to the CTW algorithm with a 1 MBytes buffer. The CTW algorithm that stores the context symbols in the leaves performs relatively poor, because in this case the number of symbols it can store in each leaf is relatively small compared to the maximum depth. As a result, its performance is comparable to the CTW algorithm that uses only a 16 kBytes buffer.

Table 4.7: The performance of the four settings for depth $D = 6$.

	length of file buffer			
	1 MBytes bytes	no bytes	$< 1/3$ bytes	16 kBytes bytes
book2	141900	141972	142197	142599
geo	58179	58185	58226	58263
obj2	71760	71807	71749	71561
paper1	14881	14896	14930	14930
pic	49705	49695	49685	49654

The CTW algorithm that stores the context symbols in the leaves has in general a more moderate growth of the number of records than any of the three algorithms that use a (limited) file buffer. This has already been observed in the plots in Figure 4.8, and it is now confirmed by the results in Table 4.9. This table gives for each entry in Table 4.8 the number of created records: the CTW algorithm without file buffer creates on three files considerably less records than the two CTW algorithms with a limited file buffer. A moderate growth can both be a

Table 4.8: The performance of the four settings for depth $D = 10$.

	length of file buffer			
	1 MBytes bytes	no bytes	$< 1/3$ bytes	16 kBytes bytes
book2	139498	140090	139720	140269
geo	58158	58174	58207	58243
obj2	70387	71151	70453	70442
paper1	14790	14835	14830	14830
pic	49142	49129	49118	49064

Table 4.9: The number of records used by each of the four settings for depth $D = 10$.

	length of file buffer			
	1 MBytes #records	no #records	$< 1/3$ #records	16 kBytes #records
book2	5554551	4566787	5147183	4637816
geo	1185940	1163866	1055889	965817
obj2	2145906	1457375	2065220	1982850
paper1	614874	456259	574978	574978
pic	930563	864167	886441	810029

disadvantage as well as an advantage. It is a disadvantage if the hash table is large enough to contain all records, because then the restricted growth can cause a loss in performance (see Table 4.8). But it can be an advantage if the hash table is too small to hold all records. In order to investigate this, each of the five files has been compressed with a too small hash table. For the files book2, geo, obj2, paper1, and pic we used hash table sizes of 16 MBytes (2796202 records), 4 MBytes (699050 records), 8 MBytes (1398101 records), 2 MBytes (349525 records), and 4 MBytes (699050 records), respectively. The results can be found in Table 4.10. It is clear that under these circumstances the CTW algorithm without file buffer performs better than the CTW algorithms with file buffer.

We decided to use the CTW with a limited file buffer for two reasons. First, experiments show that our hash table of 32 MBytes is large enough for compressing all files from the Calgary corpus with depth $D = 10$, with the exception of book1 (see also the next subsection). Therefore, we can safely use our CTW algorithm with such deep context trees, and in that case the best performance is achieved by the CTW algorithm with a file buffer. Secondly, the companion algorithm (see Chapter 7) can also use this file buffer to achieve a higher performance. On the other hand, the CTW algorithm will now use slightly more than 32 MBytes of memory, and it will fill the hash table faster. The latter problem can be eliminated by an adaptive control of the maximum depth of the context trees, but this has not been investigated.

Table 4.10: The performance if the hash table is too small.

	length of file buffer			
	1 MBytes bytes	no bytes	$< 1/3$ bytes	16 kBytes bytes
book2	146887	144873	146760	145483
geo	58513	58521	58488	58461
obj2	76321	71596	76362	75603
paper1	16353	15874	16336	16336
pic	49332	49255	49305	49172

4.5.5 EVALUATION OF THE DESIGN

For the final experiments, we use the reference set-up combined with a file buffer of 512 kBytes and a hash table of 32 MBytes. The performance of the CTW algorithm with depths $D = 8$ and $D = 10$ is compared in Table 4.11 with the results from the "state selection" algorithm [10], a well-documented PPM-variation, and PPMZ-2 v0.7. The two results marked with a * have been achieved by preprocessing the file, and will not be considered for the comparison. CTW with depth 10, is slightly better than CTW with depth 8, except on book1 for which the hash table is too small for depth 10. CTW with $D = 10$ compares favourably to "state selection". Only on the source code progp its performance is insufficient. Compared to PPMZ-2 v0.7, the results are mixed. On three of the four text files, book1, book2, and paper2 CTW gives a better performance and on text file paper1, it is slightly worse. But on almost all other files PPMZ-2 performs much better. On files bib, news, obj2, progl, progp, and trans PPMZ outperforms CTW by 0.04 bits/symbol or more. Therefore, CTW fulfils the strong performance criterion for text files, but not on the other files. But even worse, on the entire corpus, minus geo and pic, PPMZ-2 has a better performance than CTW. Thus this CTW implementation doesn't meet the weak performance criterion yet.

The complexity of the CTW algorithm is summarized in Table 4.12. The number of created records of 6 bytes each is listed in column "#records", the hash table can hold up to 5592405 records. For each file, the compression time in this table is the average time in seconds, measured over six runs. The compression speed is presented in kBytes/second (column kB/sec). Not surprisingly, CTW with depth 8 is faster than CTW with depth 10. But, from this table it is also obvious that the CTW algorithm does not meet its speed requirement yet. The poor speed and performance of CTW with $D = 10$ on book1 can be explained by the fact that it tries to create more records than possible.

In summary, the CTW algorithm uses a hash table of 32 MBytes and a file buffer of 512kBytes. Thus it meets the memory criterion. Although, it meets the strong performance criterion on the text files, it does not meet the weak performance criterion on the entire corpus, nor the speed criterion.

Table 4.11: The performance of CTW.

	CTW				"state selection"		PPMZ-2	
	$D = 8$		$D = 10$		(PPM)		v0.7	
	bytes	bits/sym	bytes	bits/sym	bits/sym	diff	bits/sym	diff
bib	24608	1.769	24529	1.764	1.788	+0.024	**1.717**	-0.047
book1	206996	**2.154**	208009	2.165	2.205	+0.040	2.195	+0.030
book2	140079	1.835	139784	**1.831**	1.876	+0.045	1.846	+0.015
geo	58160	4.544	58158	4.544	**4.508**	-0.036	4.097*	—
news	107227	2.275	106988	2.270	2.292	+0.022	**2.205**	-0.065
obj1	9835	3.659	9829	**3.657**	3.699	+0.042	3.661	+0.004
obj2	70635	2.290	70387	2.282	2.272	-0.010	**2.241**	-0.041
paper1	14804	2.228	14790	2.226	2.249	+0.023	**2.214**	-0.012
paper2	22472	2.187	22444	**2.184**	2.221	+0.037	2.185	+0.001
pic	49611	0.773	49143	**0.766**	0.795	+0.029	0.480*	—
progc	11220	2.266	11203	2.263	2.296	+0.033	**2.258**	-0.005
progl	13710	1.531	13609	1.520	1.528	+0.008	**1.445**	-0.075
progp	9801	1.588	9692	1.570	1.505	-0.065	**1.450**	-0.120
trans	15393	1.314	15202	1.298	1.291	-0.007	**1.214**	-0.084

Table 4.12: The complexity of CTW.

	CTW, $D = 8$			CTW, $D = 10$		
	#records	sec.	kB/sec	#records	sec.	kB/sec
bib	731858	14.12	7.7	959572	16.18	6.7
book1	5543214	118.19	6.4	5592395	182.00	4.1
book2	3844725	87.33	6.8	5404452	108.75	5.5
geo	1181607	10.08	9.9	1185940	10.10	9.9
news	2895810	42.87	8.6	3632393	48.02	7.7
obj1	185417	3.57	5.8	202579	3.74	5.7
obj2	1731273	26.81	9.0	2145893	28.56	8.4
paper1	490472	7.32	7.1	614874	7.88	6.6
paper2	753455	11.39	7.0	967769	12.19	6.6
pic	809575	49.60	10.1	930456	58.18	8.6
progc	357107	5.93	6.6	443959	6.14	6.3
progl	400663	9.03	7.8	529739	9.86	7.1
progp	286395	6.82	7.1	374989	7.68	6.3
trans	405943	11.18	8.2	537096	11.89	7.7

5

SUBSYSTEM 3: THE FORWARD DECOMPOSITION

This chapter discusses the implementation of a forward decomposition. The forward decomposition translates the 256-ary source symbols into bits. By using the Huffman code to determine the depths of the source symbols in the decomposition tree of the forward decomposition, the memory and computational complexity have been reduced by 30 % and 20 %, respectively. By ordering the symbols in the leaves of the decomposition tree based on their initial numbering, the performance of the CTW algorithm has improved by 1 %. The resulting design meets the speed, the memory, and the weak performance criteria.

5.1 DESCRIPTION

THIS chapter is concerned with the design of a special forward decomposition. This is an extra pass run before the CTW algorithm encodes the source sequence. It analyses the source sequence, computes a forward decomposition, and encodes the forward decomposition for the decoder. The final implementation has to fulfil the following requirements:

- The forward decomposition should not result in a loss in performance compared to the CTW algorithm with an ASCII decomposition, even though the forward decomposition has to be described too.
- The forward decomposition should increase the compression speed by about 40 % to achieve the speed criterion.
- The forward decomposition should decrease the memory complexity for `book1` by about 30 %, such that the 32 MBytes hash table is large enough for all files of the Calgary corpus.

Four different forward decompositions will be described, each based on the Huffman-code. The performance of these decompositions will be measured on the Calgary corpus, and the complexity reduction will be investigated.

5.2 BASIC ALGORITHM

5.2.1 INVESTIGATING THE FORWARD DECOMPOSITION

A forward decomposition is a binary decomposition tree (see also Section 4.3.1), in which each possible source symbol corresponds to a leaf. A symbol is coded by coding the binary decision in each node of the forward decomposition on the path from the root node to the leaf corresponding to that symbol. For this purpose, each node in the forward decomposition has its own context tree, which is used by a CTW algorithm to code the binary decision (see an example for a ternary source in Figure 5.1(b)).

The computational complexity is proportional to the average number of updated context trees per source symbol and the same holds for the memory complexity, because the context trees are grown dynamically (on average every context-tree update results in roughly one new record). Thus, these two complexities can be minimized simultaneously by minimizing the average number of bits in which the forward decomposition represents a source symbol. This can be realized by using the well-known Huffman code.

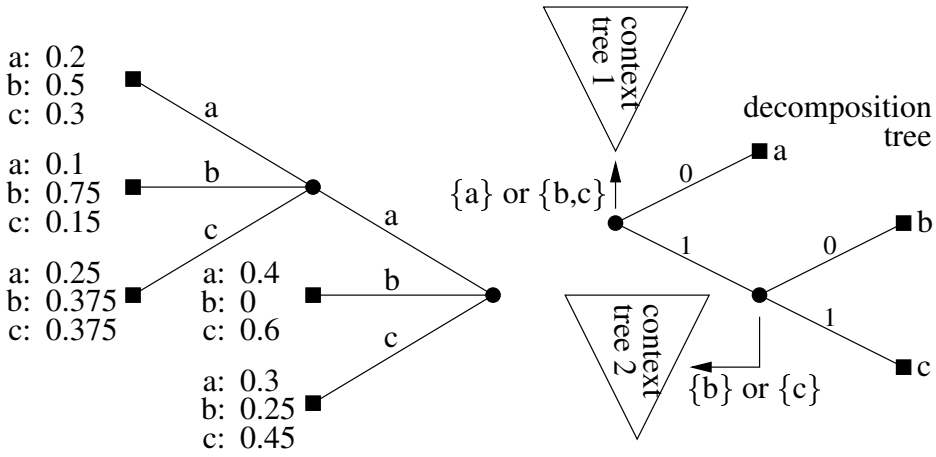
The performance of the decomposed CTW algorithm is not only determined by the average number of context tree updates. If coding would not induce model and parameter cost, then each decomposition would result in the same performance. Suppose a symbol x_n is decomposed in some M bits $z_1, \ldots, z_M$. According to Bayes' rule,

$$\begin{aligned} P(X_n = x_n | x_1^{n-1}) &= P(X_n = z_1, z_2, \ldots, z_M | x_1^{n-1}) \\ &= P(z_1 | x_1^{n-1}) \cdot P(z_2 | z_1, x_1^{n-1}) \cdot P(z_M | z_1^{M-1}, x_1^{n-1}), \end{aligned}$$

for any decomposition $z_1, \ldots, z_M$, thus every forward decomposition would perform the same. But, the CTW algorithm results in both model redundancy and parameter redundancy. The model redundancy is fixed, while the parameter redundancy grows with the length of the source sequence, thus the latter will be the dominant term for long source sequences. The number of parameters that the CTW algorithm will use, depends on the model to which it has to converge in each context tree. These models are determined by the exact splitting of the (sub)set of possible source symbols in each node of the forward decomposition. This is controlled by the grouping of the symbols in the forward decomposition, if we assume that their depths have already been fixed. It is not clear how one can determine in advance which grouping will result in the smallest models.

Here, the problem of finding a good forward decomposition is split into two parts. First the depths of the leaves of the forward decomposition are determined to minimize the number of context tree updates. Next, the leaves will be ordered according to some strategy. The simple constructions described here will not result in the optimal forward decomposition. Note, that the

main goal of this construction is to minimize the complexities. If the main goal would have been to increase the performance, a different strategy might be preferable.



(a) A ternary tree source. (b) A forward decomposition.

Figure 5.1: A ternary tree source and a forward decomposition.

Example: Figure 5.1(a) shows an example of a ternary tree source. Figure 5.2 shows two (out of three) possible forward decompositions for a ternary source and it shows for each node of the forward decomposition the model towards which the CTW algorithm has to converge. A CTW algorithm with a ternary estimator would not need a forward decomposition, and it would converge to the original tree source.

With the forward decomposition of Figure 5.2(a) the CTW has to converge to a model with the same structure as the source in both context trees. Therefore, the model cost will be doubled, compared to the CTW algorithm with a ternary estimator. The CTW algorithm with the forward decomposition uses in each leaf a binary estimator, instead of a ternary estimator. Let us for the moment assume that the parameter redundancy of a ternary estimator is roughly equal to the parameter redundancy of two binary estimators[1]. In that case, the number of parameters used by this decomposition would be equivalent to the number of parameters used by a ternary estimator. So, this decomposition would result in more model cost and about the same parameter redundancy. But, because the parameter redundancy will be dominant for sufficiently long sequences, for such sequences a CTW algorithm based on this decomposition would perform comparable to one that uses a ternary estimator.

If the forward decomposition in Figure 5.2(b) is used, the model required to estimate the probability distribution of symbols "a" and "c" (context tree 2) collapses to the memoryless model! With this forward decomposition the CTW algorithm has still (slightly) more model cost, but the number of necessary parameters is significantly reduced from ten to six. Therefore, already for short sequences, the CTW algorithm with this forward decomposition will result in a superior performance compared to a ternary CTW algorithm.

[1]We base this rule of thumb on Rissanen's converse (Section 1.5.7 already gave the exact definition for binary sources). Consider a multi-alphabet $\mathcal{X}$ with $|\mathcal{X}|$ symbols. The measure of vectors Θ over this alphabet for which the parameter redundancy is smaller than $\frac{|\mathcal{X}|-1}{2}\log_2(N)$ (the constant terms have been omitted) goes to zero for infinitely long sequences. Thus asymptotically, the redundancy of an estimator for a $|\mathcal{X}|$-ary alphabet is comparable to the redundancy of $|\mathcal{X}|-1$ binary estimators.

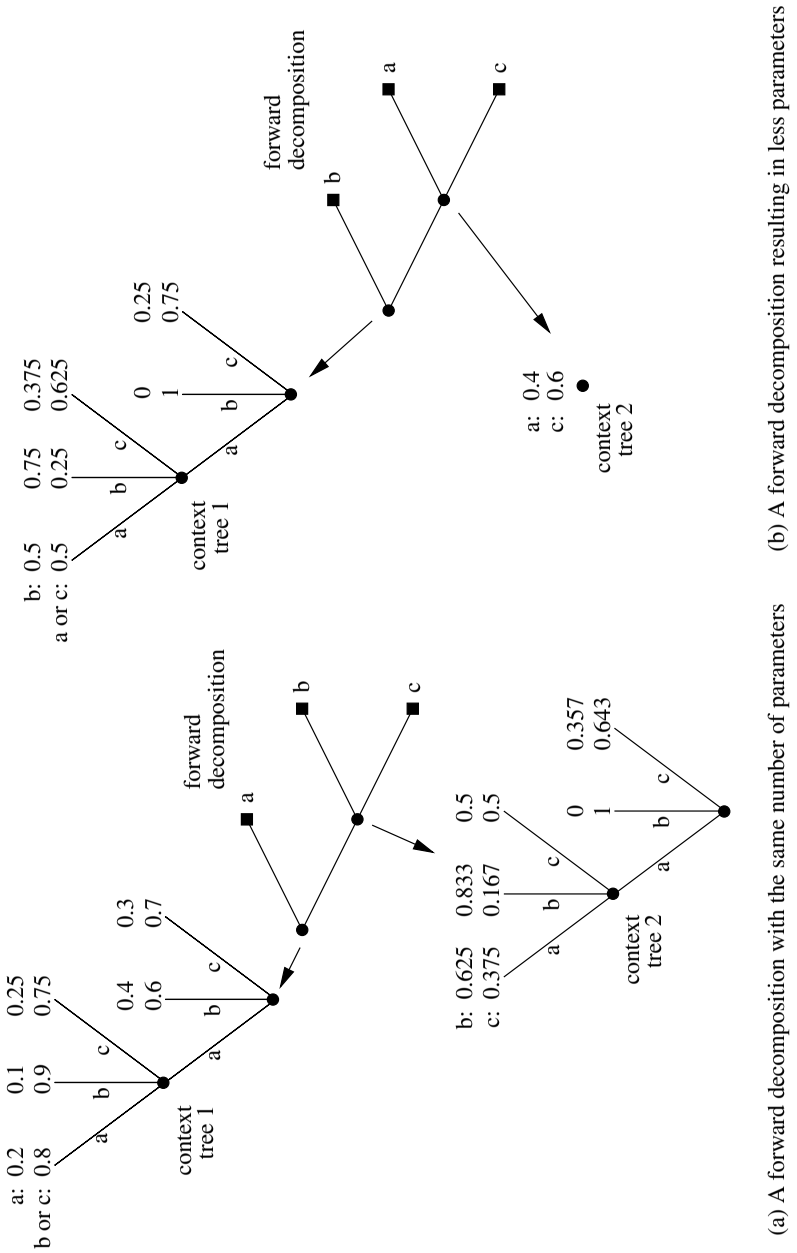


Figure 5.2: Two different forward decompositions for the same source.

Even though the example is artificial, it shows two interesting properties. First, a forward decomposition will result in higher model redundancy, but the parameter redundancy will be similar or lower. Secondly, in the example, the forward decomposition in Figure 5.2(b) is the only one of the three possible decompositions that results in a decrease in parameter redundancy. But, in a source sequence generated by the given source, symbol "c" will be the most probable symbol (the probability distribution of the symbols is (0.29,0.27,0.44)). This shows that our simple construction will indeed reduce the complexity, but may not achieve the optimal performance.

In practice, the source sequence has a limited length. As a result, the counts in the context trees will only approximate the symbol probabilities as defined by the actual model. Therefore, even if the symbol probabilities in the actual source are such that a model in a context tree will not collapse to a smaller model, in practice, this can still happen due to a small number of counts.

5.2.2 COMPUTING THE SYMBOL DEPTHS

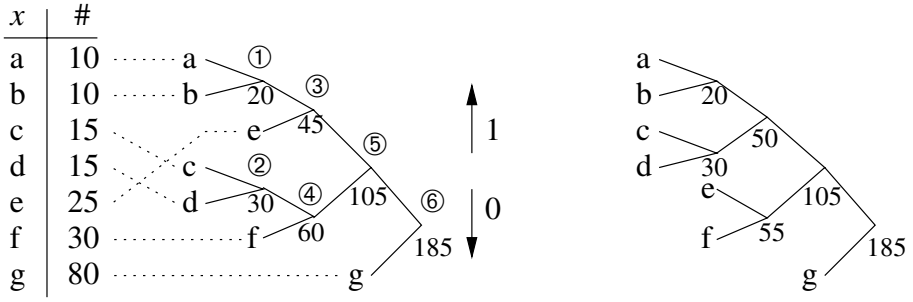
Our decomposition will be based on the Huffman code.

Theorem 5.1 (Huffman code [19]) *Consider a source alphabet $\mathcal{X}$ with a known probability distribution $P(x)$. The Huffman code is a prefix code with code words of length $l_H(x)$ for all symbols $x \in \mathcal{X}$. Then, for any other prefix code, with lengths $l(x)$ for all $x \in \mathcal{X}$, the following holds:*

$$L_H = \sum_{x \in \mathcal{X}} P(x) l_H(x) \leq \sum_{x \in \mathcal{X}} P(x) l(x) = L.$$

The Huffman code achieves the shortest average code word length.

The construction of the Huffman code is very simple. First, compute the probabilities or the number of occurrences of the source symbols. Next, combine the two least probable symbols in one "super" symbol which has as frequency the sum of both frequencies, and repeat this step until there is only one super symbol left.



(a) The Huffman construction (b) An equivalent Huffman tree

Figure 5.3: Example of the construction of a Huffman code.

Example: Suppose the source sequence generated by a 7-ary source has to be encoded. The table in Figure 5.3(a) gives the number of occurrences of each of the 7 source symbols. Here, the Huffman construction will construct the code tree in six steps (see the circled numbers in the tree). First, symbols

"a" and "b" are combined into one super symbol with 20 occurrences. In the code tree this is represented by the node that has "a" and "b" as children. Next, symbols "c" and "d" are combined in the same way. In the third step, there are five symbols left: "a or b" with 20 occurrences, "e" with 25, "c or d" with 30, "f" with 30, and "g" with 80. Thus, symbol "e" will now be combined with super symbol "a or b" and node number ③ is created. This process continues until there is one code tree with all symbols. From this code tree the Huffman code can be found by assigning a zero to one branch of a node and a one to the other branch. For example, if we assign a zero to the downward branches and a one to the upward branches, then symbol "c" gets code word 1011.

The Huffman code minimizes the average code word length, thus if its code tree is used as forward decomposition, the average number of context tree updates is minimized too.

5.2.3 COMPUTING THE SYMBOL ORDER

Every code tree which places the source symbols at the same depth as the Huffman code, will achieve the same average code word length. These code trees are called equivalent Huffman trees (an example is shown in Figure 5.3(b)). It is not clear which equivalent Huffman tree has to be used as forward decomposition to achieve the shortest code word length with the CTW algorithm.

Previous experiments [25, 48] showed that related symbols should be placed in one subtree of the forward decomposition. Especially decompositions for texts were investigated, and the best decompositions used separate subtrees for e.g. numbers, lower case letters, and capital letters, in which these last two groups are split again in a subtree for vowels and one for consonants, etc. Such a decomposition creates a hierarchy in the decisions: global decisions (capital or lower case, vowel or consonant, etc.) should go first. This also agrees with the intuitive idea: symbols that are almost interchangeable (like the "a" and "e" in English), should be kept together as long as possible, because these two will be almost indistinguishable compared to the other symbols, and it has no use complicating earlier decisions by splitting these two in advance.

It is very difficult to deduce which symbols are most interchangeable, and should be grouped together. Here, we will assume that the initial 8 bits representation of the symbols is chosen such that related symbols are already grouped together and their numbers cover a small range. This is e.g. true for the ASCII code, in which the group of lower case letters, the group of upper case letters, and the group of numbers, each are numbered consecutively. Our special forward decompositions will use this assumption by trying to keep symbols with numbers in the same range together. The results in Section 5.4 validate this assumption, because the straightforward ASCII decomposition performs better on 11 out of 14 files than the Huffman decomposition (the code tree of the Huffman code, which completely mixes the symbols). The three exceptions are `book1`, for which the hash table is too small for the ASCII decomposition, and `geo` (4 columns of 32 bits numbers) and `pic` (a picture with only *one* bit plane), in which the initial numbering of the symbols reveals nothing about the structure of the data.

We have experimented with several different orders of the symbols in the forward decomposition. The forward decomposition based on the code tree of the Huffman code is denoted by **Huf-I**. Here, we will mention three alternative strategies (from [58]):

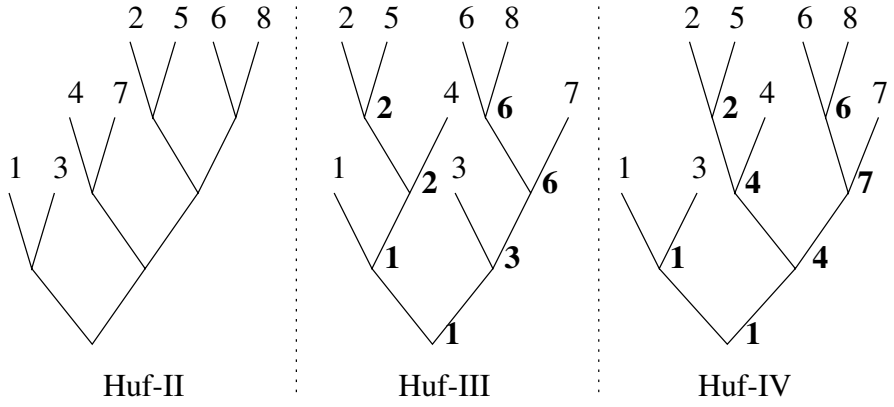


Figure 5.4: An example of the decomposition trees of the three strategies.

Huf-II: First sort the symbols according to their depth in the Huffman tree, and then for each depth sort the symbols on their initial number.

Huf-III: Combine the two symbols at the highest depth which have the highest numbers. The depth of the resulting super symbol is one level lower than the depths of both symbols, and its number is the lowest number of all symbols collected in this super symbol. This process is continued until only one super symbol at depth 0 remains.

Huf-IV: Combine the two symbols at the highest depth which have the highest numbers. The depth of the resulting super symbol is one level lower than the depths of both symbols, and its number is now the lowest symbol number among all symbols (*not* super symbols) that are at the lowest depth inside the subtree of this super symbol. Repeat this step until all symbols are collected in one tree.

Note that the Huf-II, Huf-III and Huf-IV decompositions introduce an undesirable property: because these decompositions depend on the initial numbering of the source symbols, a permutation of the source symbols is going to influence the performance of the program.

Example: Suppose an 8-ary source generates a source sequence. The Huffman tree has the symbols 2, 5, 6 and 8 at depth 4, the symbols 4 and 7 at depth 3 and the symbols 1 and 3 at depth 2. Figure 5.4 shows the resulting forward decompositions for each of the three strategies. The bold numbers at the nodes indicate the number assigned to the super symbol represented by the subtree from that node. The Huf-II strategy results in "good" pairs in general, but the larger subtrees often contain a wide range of values, e.g., its right most subtree has symbols with numbers 2, 5, 6 and 8. The other two strategies create better subtrees, in which the symbol numbers cover a much smaller range. Both the Huf-III decomposition and the Huf-IV decomposition create two subtrees of three leaves with symbol numbers 2, 4 and 5 in one subtree and 6, 7 and 8 in the other.

5.3 IMPLEMENTATION: DESCRIBING THE FORWARD DECOMPOSITION

Implementing the special forward decomposition is simple. The encoder first counts the source symbol frequencies, and it determines the forward decomposition, with one of the four strategies. Next, it describes the forward decomposition for the decoder, and it starts encoding the sequence. The decoder first reads the description of the forward decomposition, and it can start decoding. The only implementation issue is the description of the forward decomposition. The symbols in the Huf-I decomposition are not ordered, thus it has to be described completely for the decoder, while for the other three decompositions only the depths of the symbols have to described.

For the Huf-I decomposition the encoder has to describe the entire forward decomposition. It can describe the code tree of the Huffman code in two parts. First it describes the tree structure by walking depth-first through the tree and encoding each new node on the path with a 0 and each leaf on the path with a 1. Next, it encodes the symbols in the leaves (with 8 bits) in the same order as in which they were visited. If the Huffman tree has $|S|$ leaves, then the entire description (structure plus symbols) will cost $(2|S|-1)+(8|S|) = 10|S|-1$ bits. Note that the description of the tree structure has to follow some rules. For example, any prefix of this description, except for the entire description, will not contain more ones than zeros. Unfortunately, such rules cannot be used easily to shorten the length of the tree description.

Example: The Huffman tree in Figure 5.3(a) will be encoded by (we take the lower branches first) 0 1 0 0 1 0 1 1 0 1 0 1 1 followed by the encoding of the leaves: g, f, d, c, e, b, a.

The encoder uses only the Huffman depth of each symbol, and the initial numbering to compute the Huf-II, Huf-III and Huf-IV decompositions. Therefore, it only needs to describe the Huffman depths of all symbols, after which the decoder is able to reconstruct the forward decomposition. If we limit the depth of the forward decomposition to 31 levels, it takes not more than 5 bits to specify a depth (depth 0 indicates that a symbol did not occur). Thus the depths of all symbols can be encoded in $256 \cdot 5$ bits, or 160 bytes. One question remains. What is the minimum number of source symbols that could result in a Huffman tree of depth 32?

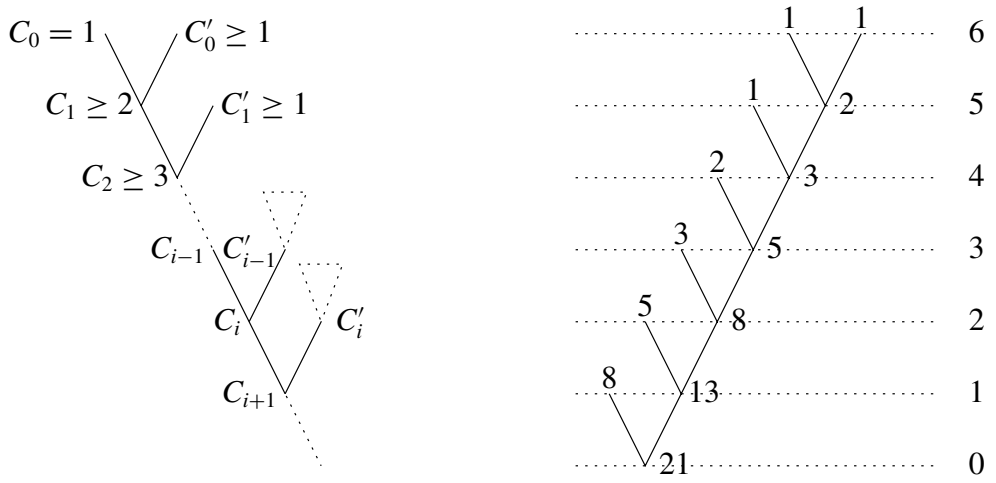
Theorem 5.2 (Maximum depth of a Huffman tree [24]) *Define the Fibonacci number f_i as the i-th number in the recursion $f_{i+1} = f_i + f_{i-1}$ for $i \geq 2$, with $f_1 = 1$ and $f_2 = 1$. The maximum depth D_H of a Huffman tree constructed from a source sequence x_1^N, with $f_{i'} \leq N < f_{i'+1}$, is upper bounded by,*

$$D_H \leq i' - 2,$$

and this bound is achievable.

This theorem is slightly sharper, but also less general, than the one given in [24], because it uses the fact that the least probable source symbol occurs at least once, and thus its probability p is at least $p \geq \frac{1}{N}$. For sake of completeness a short proof is provided.

Proof: The least probable symbol in the source sequence occurs at least once and will be placed at the maximum depth of the Huffman tree D_H. Consider the path in the Huffman tree formed by the nodes from the root node to the leaf representing this symbol. Let us denote the number of counts in these nodes by C_i, with $i = 0, 1, \ldots, D_H$, in which $C_0 \geq 1$ are the counts in the leaf



(a) Dependencies in the Huffman tree (b) A "worst case" Huffman tree

Figure 5.5: The minimal number of symbols necessary to create a Huffman tree of a certain depth.

(see Figure 5.5(a)). Each node on this path has also a child which is not on the path, and with C' we denote the counts in that child. From the nature of the Huffman construction we find:

$$C'_i \geq C_{i-1} \tag{5.1}$$

(Lemma 1 equation (1) in [24]). If (5.1) would not hold, then these two subtrees could be exchanged and the average code word length would decrease. We will now prove by induction that

$$C_i \geq f_{i+2}, \tag{5.2}$$

for $i \geq 2$.

For $i = 2$, $C_2 = C_1 + C'_1$. A Huffman tree of depth one should have at least two counts, one in each of its two children, so $C_1 \geq 2 = f_3$ consequently $C_2 \geq f_3 + C'_1 \geq f_3 + C_0 \geq f_3 + f_2 = f_4$ (in which the second inequality follows from (5.1)).

Suppose (5.2) holds for some i then for $i + 1$, $C_{i+1} = C_i + C'_i \geq C_i + C_{i-1} \geq f_{i+2} + f_{i+1} = f_{i+3}$, in which the induction hypothesis (5.2) is used in the second inequality. This proves (5.2).

The least probable symbol will have at least one count and will be placed furthest away from the root node in the Huffman tree. If it is placed at some depth D_H from the root node, then the root node has to have at least $C_{D_H} \geq f_{D_H+2}$ counts. In other words, there should be at least $N \geq f_{D_H+2}$ counts for a Huffman tree of depth D_H, which proves the inequality in the theorem. A source sequence of length f_{D_H+2} with $D_H + 1$ different source symbols, which occur 1, f_1, f_2, ..., f_{D_H} times, satisfies all inequalities with equality, and also achieves the bound with equality (an example of such a Huffman tree is shown in Figure 5.5(b) for depth six[2]). □

[2]Note that in this case a different Huffman tree can be constructed that has maximum depth five.

Our Huffman tree can have a maximum depth of 31. It would require at least a source sequence of length $f_{34} = 5,702,887$ symbols to construct a deeper Huffman tree. In practice it is very unlikely that sequences of this length, or even much longer sequences, result in such a tree. Such problems could be resolved by scaling the counts of the symbols such that their sum is less than 5,702,887 before the Huffman depths are computed. But this is not implemented.

The Huf-II, Huf-III and Huf-IV decompositions can be described in 160 bytes, but more efficient schemes are possible. One could e.g. use a Huffman code to describe the depths of the symbols. We examined one other alternative scheme. It encodes for each depth, starting at the lowest depth, all symbols at that depth. It starts at depth zero. It sends a "1" to increase the depth by one, and it sends a "0" followed by a symbol (encoded in 8 bits) to place a symbol at the current depth. The decoder knows that the description is complete when all free slots at the current depth are occupied by symbols. This scheme costs 9 bits per source symbol plus D_H bits, in which D_H is the depth of the Huffman tree. It is more efficient than the scheme with a fixed cost of 160 bytes for files with up to about 140 different source symbols.

Example: Consider the decomposition trees in Figure 5.4. This code describes the depths of the symbols as following: 1, 1, (0, "1"), (0, "3"), 1, (0, "4"), (0, "7"), 1, (0, "2"), (0, "5"), (0, "6"), (0, "8").

Our final design can use either this alternative description, or store the depths in 160 bytes. It simply computes which one is more efficient, and it uses one bit to specify the chosen description for the decoder.

5.4 FIRST EVALUATION

The performance of the CTW algorithm has been compared for five different decompositions: the ASCII decomposition, and the special Huf-I to Huf-IV decompositions. Experiments show that the reference set-up also results in the best performance for each decomposition: $D = 10$, β is bounded to $[-2500, 2500]$, $\alpha = 16$, the file buffer has size 512 kBytes, and the hash table is 32 MBytes. The sizes in bytes of the compressed files of the Calgary corpus are collected in Table 5.1. These sizes include the costs of the descriptions of the forward decompositions. The Huf-I decomposition improves the results only on three files as explained before. This clearly shows that there is a penalty for not grouping the symbols. The two strategies that try hardest to find a good grouping of the symbols, Huf-III and Huf-IV, perform best. They both gain about 8 kBytes (about 0.02 bits/symbol, or 1 % in compressed size) on the Calgary corpus compared to the ASCII decomposition. The performance of CTW algorithms with these two decompositions differ only marginally on the files of the Calgary corpus, but also on the files of the Canterbury corpus and for long text files. We choose to use the Huf-IV decomposition, for the tiny extra gain in performance.

The actual goal of a specific decomposition is the reduction of complexity, and the increase of compression speed. In Table 5.2 the complexity of the CTW algorithm with the ASCII decomposition is compared to the one with the Huf-IV decomposition. For each file and each decomposition, the number of used records, and the compression time are given. The time has been measured over six runs. The Huf-IV decomposition reduces the number of necessary records by 30 to 40 %, except on the two object files. It reduces the number of records for `book1` by more

Table 5.1: The performance of the CTW algorithm with different decompositions.

file	ASCII	Huf-I	Huf-II	Huf-III	Huf-IV	PPMZ-2
bib	24529	25013	24654	24466	24441	**23873**
book1	208009	207802	207506	**206045**	206151	210952
book2	139784	141538	141036	139461	**139222**	140932
geo	58158	56997	**54444**	56646	56541	52446*
news	106988	107706	107485	106190	106221	**103951**
obj1	9829	10174	9845	**9748**	9783	9841
obj2	70387	72404	70723	69513	69511	**69137**
paper1	14790	14991	14936	14704	**14696**	14711
paper2	22444	22638	22540	**22300**	22355	22449
pic	49143	47746	47451	47239	**47194**	30814*
progc	11203	11470	11340	**11177**	11209	11178
progl	13609	13758	13632	13579	13588	**12938**
progp	9692	9879	9696	9561	9558	**8948**
trans	15202	15533	15395	15191	15210	**14224**
total	753767	757649	750683	745820	745680	**726400**

Table 5.2: A comparison between the two decompositions

file	ASCII dec.		Huf-IV decomposition				
	records	time	records		time		speed
	#	sec.	#	%	sec.	%	kB/s
bib	959572	16.18	632512	-34.1	11.00	-32.0	9.9
book1	5592395	182.00	4815707	-13.9	67.43	-63.0	11.1
book2	5404452	108.75	3407626	-36.9	55.16	-49.3	10.8
geo	1185940	10.10	809168	-31.8	8.30	-17.8	12.0
news	3632393	48.02	2370655	-34.7	31.06	-35.3	11.9
obj1	202579	3.74	169441	-16.4	3.75	+0.0	5.6
obj2	2145893	28.56	1721116	-19.8	24.35	-14.7	9.9
paper1	614874	7.88	387356	-37.0	6.29	-20.2	8.3
paper2	967769	12.19	565403	-41.6	8.00	-34.4	10.0
pic	930456	58.18	494480	-46.9	14.00	-75.9	35.8
progc	443959	6.14	292023	-34.2	5.19	-15.5	7.5
progl	529739	9.86	324971	-38.7	7.14	-27.6	9.8
progp	374989	7.68	246435	-34.3	5.90	-23.3	8.2
trans	537096	11.89	379965	-29.3	9.70	-18.4	9.4
total	23522106	511.17	16616858	-29.4	257.27	-49.7	11.9

than 40 %, such that now all records fit in the hash table. On most files the total compression time has been reduced by more than 20 % (so the compression speed, given in kBytes/second, increases by 25 %), except on the smaller files, for which the initialization of the hash table costs relatively much time. On book1 and pic the speed increases tremendously. On book1 the hash table is too small for the ASCII decomposition, which has a severe impact on the search speed. In the file pic symbol 0 occurs very often and is placed at depth 1 in the Huffman decompositions. The overall compression speed has increased by more than 40 % to about 11.9 kBytes/second. Furthermore, the Huf-IV decomposition also improves the performance of the CTW algorithm significantly. It compresses the Calgary corpus (minus geo and pic which are preprocessed by PPMZ-2) to 641945 bytes. PPMZ-2 v0.7 compresses these 12 files to 643140 bytes. In general, the CTW algorithm has a better performance on the four text files book1, book2, paper1, and paper2 (382424 bytes versus 389044 bytes) and PPMZ-2 on the other files.

In summary, the advantages of this forward decomposition clearly outweigh its disadvantages (a two-pass scheme, influence of the initial numbering of the symbols). With this extension the design meets the memory requirement, the speed requirement, the weak performance requirement, and the strong performance requirement on the text files (but not on the other files).

6

SUBSYSTEM 4: THE SNAKE ALGORITHM

This chapter discusses the implementation of a combining algorithm called the snake algorithm. This algorithm is a greedy implementation of the switching method, thus it will be able to switch at any time from one of the two data compression algorithms to the other one. It causes a negligible increase in memory complexity, and about a 10 % reduction in compression speed. Because it is an approximation of the switching method, its exact switching behaviour is undetermined. Its performance will be evaluated in Chapter 7.

6.1 DESCRIPTION

THIS chapter is concerned with the design of the snake algorithm. The CTW algorithm and the companion algorithm both estimate the probability distribution of the next bit. The switching method provides a framework with which, at each moment, the estimate that performs locally best can be used. The switching algorithm is a comprehensive implementation of the switching method, based on weighting. The snake algorithm is a less accurate implementation of the switching method, but it has a much lower complexity. For this reason, we will use the snake algorithm in our final design. Because the snake algorithm will only be used together with the companion algorithm, the requirements for the snake algorithm cannot be separated from those of the companion algorithm.

- The snake algorithm together with the companion algorithm, should improve the performance of CTW on non-text files, such that the performance is now within 0.04 bits/symbol of the best competitive algorithms.
- The snake algorithm should result in only a small increase in memory and computational complexity.

First the switching method will be discussed, followed by the two implementations: the switching algorithm and the snake algorithm. Next, the implementation details of the snake algorithm are described and its influence on the compression speed is measured. The last section of this chapter discusses the workings of the snake algorithm.

6.2 BASIC ALGORITHM

6.2.1 THE SWITCHING METHOD

The switching method and the switching algorithm have been published in [56] and they are closely related to work in [59]. The switching method defines a way in which two universal source coding algorithms can be combined. Consider a source sequence $x_1, \ldots, x_N$. Suppose that two sequential modelling algorithms, $\mathcal{A}$ and $\mathcal{B}$, run both along the entire source sequence, and give for every symbol an estimate of its probability distribution. At each moment the encoder in the switching method uses the estimate from only one of the two modelling algorithms to encode the next source symbol. The estimate of the other modelling algorithm is ignored, but the statistics of both modelling algorithms are updated (different techniques are described in [39]). The decoder has to know which algorithm was used to encode each symbol. The switching method uses a transition sequence to encode this (an alternative strategy can be found in [69]). It starts using modelling algorithm $\mathcal{A}$ and it can switch from one modelling algorithm to the other one between any two source symbols. In the transition sequence $t_1, \ldots, t_N$, $t_i = 1$ denotes that the encoder switched from one algorithm to the other one between source symbols x_{i-1} and x_i, and otherwise $t_i = 0$.

source seq.	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
	C	o	m	p	r	e	s	s	i	o
best algorithm	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$
blocks			$\mathcal{A}$			$\mathcal{B}$			$\mathcal{A}$	
transition seq.	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
	0	0	0	0	1	0	0	1	0	0

Figure 6.1: Example of a transition sequence.

Example: Figure 6.1 shows an example of a transition sequence. Suppose the first 10 symbols of the source sequence are "C o m p r e s s i o". Then for each symbol the encoder can determine which of the two algorithms assigns the highest probability to that symbol. Since the transition sequence has to be encoded too, it is in practice more efficient to find larger blocks in which one algorithm will be used, even though some symbols in such block get a higher probability from the other algorithm. Once the encoder decided which algorithm to use for each source symbol, the transition sequence can be determined. It is

zero everywhere, except at the start of a new block. In this case, a new block starts between x_4 and x_5 and between x_7 and x_8, thus $t_5 = 1$ and $t_8 = 1$.

The encoder has to encode both $t_1, \ldots, t_N$ and $x_1, \ldots, x_N$. In principle any encoding scheme can be used to encode the transition sequence. But it should not describe the entire transition sequence first, followed by the encoding of the source sequence, because then our design would require yet another pass over the source sequence (on top of the second pass for the encoder introduced by the forward decomposition). Therefore, the switching method intertwines the transition sequence with the source sequence, $t_1, x_1, \ldots, t_N, x_N$, which allows sequential encoding and decoding.

Once an encoding scheme for the transition sequence is chosen, the encoder can try to find the best transition sequence t_1^N:

$$\min_{t_1^N} l(t_1^N) + l(x_1^N | t_1^N), \tag{6.1}$$

in which $l(t_1^N)$ denotes the length of the code word describing t_1^N, and $l(x_1^N|t_1^N)$ denotes the length of the code word describing x_1^N given the transition sequence t_1^N. Solving (6.1) is not a trivial problem. Instead of trying to find the best transition sequence, it is possible to weight over all transition sequences in a relatively efficient way, by choosing the encoding of the transition sequence in a specific way. This is done by the switching algorithm.

6.2.2 THE SWITCHING ALGORITHM

The switching algorithm weights over all 2^N possible transition sequences. Thus it has to compute,

$$P_w(x_1^n) = \sum_{t_1^n} P(t_1^n) P(x_1^n | t_1^n), \tag{6.2}$$

for all $n = 1, \ldots, N$. Here $P(t_1^n)$ is the probability assigned to transition sequence t_1^n and $P(x_1^n|t_1^n)$ is the probability of the source symbols x_1^n, given the transition sequence. The switching algorithm uses the memoryless Krichevsky-Trofimov estimator (see Section 1.5.6) to encode a transition sequence. This will allow the switching algorithm to compute (6.2) efficiently. Consider the structure corresponding to the switching algorithm in Figure 6.2. Each possible transition sequence is a path through this structure. Every path starts in state $(0, 0)$. If the path passes state (n, j), then there have been j switches from one compression algorithm to the other (ones in the transition sequence) after n source symbols. Suppose the first n source symbols were encoded with j switches, which brought the encoder to state (n, j). Now source symbol x_{n+1} has to be encoded. If $t_{n+1} = 0$, then the encoder does not switch to the other compression algorithm between symbols x_n and x_{n+1}. As a result it goes to state $(n + 1, j)$, and it encodes transition symbol t_{n+1} with probability:

$$P(T_{n+1} = 0) = \frac{n - j + \frac{1}{2}}{n + 1}. \tag{6.3}$$

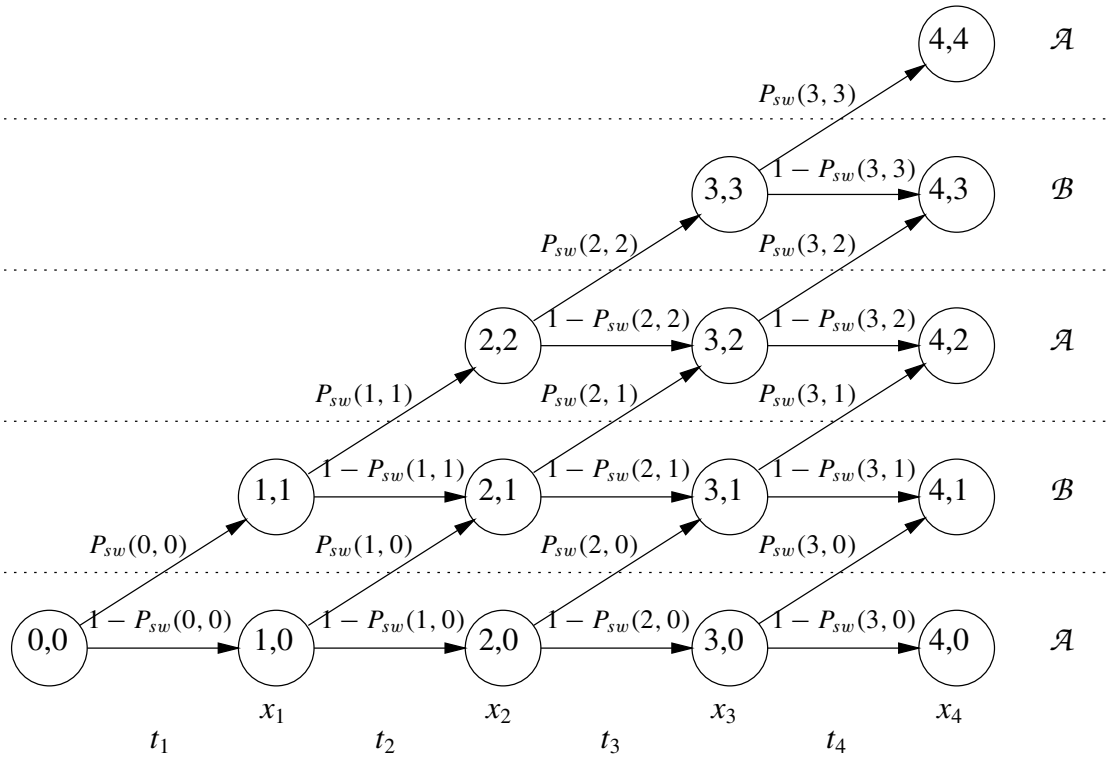


Figure 6.2: A part of the switching structure.

If $t_{n+1} = 1$ then the encoder switches to the other compression algorithm. It goes to state $(n+1, j+1)$ and it encodes transition symbol t_{n+1} with probability:

$$P_{sw}(n, j) \stackrel{\triangle}{=} P(T_{n+1} = 1) = \frac{j + \frac{1}{2}}{n+1}. \tag{6.4}$$

Now in the new state ($(n+1, j)$ or $(n+1, j+1)$) it encodes the new source symbol x_{n+1} with algorithm $\mathcal{A}$ if the number of switches so far is even, and it will use algorithm $\mathcal{B}$ if the number of switches so far is odd.

From Figure 6.2 it is clear that there is only one state (n, j), although there are many different transition sequences that have j switches after n symbols. But because the probability distribution of symbol x_n depends only on j, and the probability distribution of t_{n+1} depends only on n and j, every transition sequence that passes state (n, j) will use the same probability distributions for x_n and t_{n+1}, and thus they can all share this single state. This allows the switching algorithm to weight efficiently over all possible transition sequences. It computes in every state (n, j) the weighted probability $P_w^{n,j}(x_1^n)$: the probability of the source symbols $x_1, \ldots, x_n$ weighted over all transition sequences with j switches. State (n, j) can be reached with a switch from state $(n-1, j-1)$ and without a switch from state $(n-1, j)$. Therefore the probability $P_w^{n,j}(x_1^n)$ can be expressed in terms of probabilities $P_w^{n-1,j}(x_1^{n-1})$ and $P_w^{n-1,j-1}(x_1^{n-1})$ as follows

$$P_w^{n,j}(x_1^n) = \left[(1 - P_{sw}(n-1, j))\, P_w^{n-1,j}(x_1^{n-1}) + P_{sw}(n-1, j-1)\, P_w^{n-1,j-1}(x_1^{n-1})\right] P_{\mathcal{A}}(x_n | x_1^{n-1})$$
$$\text{if } 0 \le j \le n \text{ and } j \text{ even}, \tag{6.5}$$

$$P_w^{n,j}(x_1^n) = \left[(1 - P_{sw}(n-1, j))\, P_w^{n-1,j}(x_1^{n-1}) + P_{sw}(n-1, j-1)\, P_w^{n-1,j-1}(x_1^{n-1})\right] P_{\mathcal{B}}(x_n|x_1^{n-1})$$
$$\text{if } 0 \le j \le n \text{ and } j \text{ odd}, \quad (6.6)$$

if we define $P_w^{0,0} = 1$, $P_w^{n-1,-1} = 0$ for all n, and $P_w^{n-1,n} = 0$ for all n. The additions in (6.5) and (6.6) give the probability of reaching state (n, j), after which the new source symbol x_n still has to be encoded with either probability $P_{\mathcal{A}}(x_n|x_1^{n-1})$ or with $P_{\mathcal{B}}(x_n|x_1^{n-1})$ depending whether j is even or odd, respectively.

The weighted probability for the sequence x_1^n can then be found by adding over all $n+1$ states:

$$P_w(x_1^n) = \sum_{0 \le j \le n} P_w^{n,j}(x_1^n). \quad (6.7)$$

For a sequence of length N, we need at most $N+1$ states. Thus the memory complexity is linear, since the number of states grows linearly in the length of the sequence. The computational complexity is quadratically in the number of symbols.

The worst case performance of the switching algorithm can be upper bounded. If algorithm $\mathcal{A}$ assigns a higher probability to the entire source sequence than algorithm $\mathcal{B}$ then the extra redundancy of the switching algorithm will be at most $\frac{1}{2}\log_2 N + 1$ bit relative to algorithm $\mathcal{A}$ (this is the redundancy of the Krichevsky-Trofimov estimator for a deterministic source). If algorithm $\mathcal{B}$ gives a higher probability for this sequence then the switching algorithm has to make (at least) *one switch*, and its redundancy compared to algorithm $\mathcal{B}$ is at most $\frac{3}{2}\log_2 N + 2$ bit. In general one can prove that the cost of making K switches is upper bounded by $K \log_2 N + \frac{1}{2}\log_2 N + 2$ bit. Thus, besides the small fixed cost, switching introduces a cost of at most $\log_2 N$ bit per switch.

6.2.3 THE SNAKE ALGORITHM

The switching algorithm has the desired switching behaviour between two universal source coding algorithms: it performs better than an algorithm that computes the best transition sequence and encodes that sequence with the KT-estimator. But the memory usage of the switching algorithm grows linearly with the length of the source sequence, and the number of computations grows quadratically with the length of the source sequence. The resulting complexity problems are solved with the snake algorithm. This is a greedy combining algorithm, and it needs only two states: one for each of the two universal source coding algorithms. The structure of the snake algorithm, see Figure 6.3, is a collapsed version of the structure of the switching algorithm (Figure 6.2). We term this weighting algorithm the snake algorithm, because its structure resembles a simplified drawing of a snake.

A state $(n, \mathcal{A})$ in the snake algorithm represents all transition sequences that use algorithm $\mathcal{A}$ to encode symbol x_n. Thus in contrast with the switching algorithm, this state is reached by all transition sequences with an even number of switches. Similarly, state $(n, \mathcal{B})$ is reached by all transition sequences with an odd number of switches. As a result, the snake algorithm uses in a state only one probability distribution for the next transition symbol, even though this state is visited by transition sequences with different numbers of switches. This can degrade its performance compared to that of the switching algorithm. In order to minimize the loss in

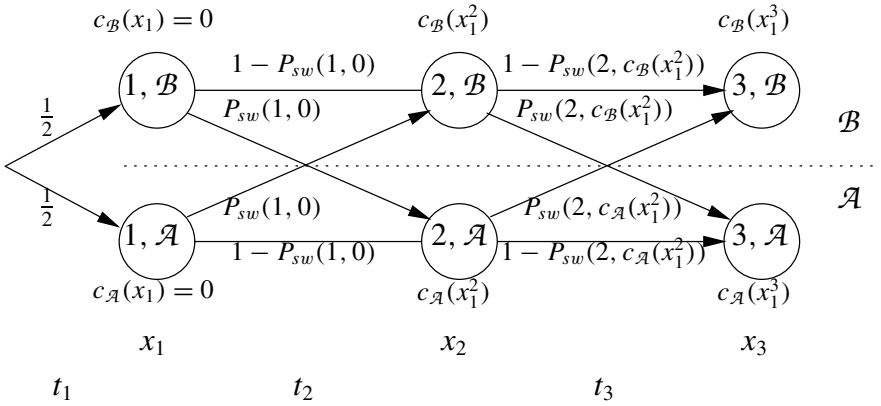


Figure 6.3: A part of the structure of the snake algorithm.

performance, the snake algorithm tries to "track" the best transition sequence. For this purpose it has a count associated with every state. The count is an estimate of the number of switches that would have occurred in the best transition sequence so far. With these counts $c_{\mathcal{A}}$ and $c_{\mathcal{B}}$ the weighted probabilities can be computed similar to (6.5) and (6.6)[1]. They are defined as:

$$P_w^{n,\mathcal{A}}(x_1^n) = \big[(1 - P_{sw}(n-1, c_{\mathcal{A}}(x_1^{n-1}))) \cdot P_w^{n-1,\mathcal{A}}(x_1^{n-1}) + P_{sw}(n-1, c_{\mathcal{B}}(x_1^{n-1}))\, P_w^{n-1,\mathcal{B}}(x_1^{n-1})\big] \cdot P_{\mathcal{A}}(x_n | x_1^{n-1}), \quad (6.8)$$

$$P_w^{n,\mathcal{B}}(x_1^n) = \big[P_{sw}(n-1, c_{\mathcal{A}}(x_1^{n-1})) \cdot P_w^{n-1,\mathcal{A}}(x_1^{n-1}) + (1 - P_{sw}(n-1, c_{\mathcal{B}}(x_1^{n-1})))\, P_w^{n-1,\mathcal{B}}(x_1^{n-1})\big] \cdot P_{\mathcal{B}}(x_n | x_1^{n-1}), \quad (6.9)$$

in which P_{sw} is defined as in (6.4). The weighted probability for the sequence so far can be found by adding $P_w^{n,\mathcal{A}}(x_1^n)$ and $P_w^{n,\mathcal{B}}(x_1^n)$. The first term of the weighted probabilities in (6.8) and (6.9) consists of two parts: the contribution from the previous state for the other algorithm followed by a switch and the probability from the previous state for the matching algorithm followed by a "non-switch". The greedy snake algorithm assumes that in each state the transition that gave the highest contribution to the probability is the best transition. This transition then determines the new count. This results in the following algorithm:

$$c_{\mathcal{A}}(x_1^n) = \begin{cases} c_{\mathcal{A}}(x_1^{n-1}) & \text{if } (1 - P_{sw}(n-1, c_{\mathcal{A}}(x_1^{n-1}))) \cdot P_w^{n-1,\mathcal{A}}(x_1^{n-1}) \\ & \quad \geq P_{sw}(n-1, c_{\mathcal{B}}(x_1^{n-1}))\, P_w^{n-1,\mathcal{B}}(x_1^{n-1}), \\ c_{\mathcal{B}}(x_1^{n-1}) + 1 & \text{if } (1 - P_{sw}(n-1, c_{\mathcal{A}}(x_1^{n-1}))) \cdot P_w^{n-1,\mathcal{A}}(x_1^{n-1}) \\ & \quad < P_{sw}(n-1, c_{\mathcal{B}}(x_1^{n-1}))\, P_w^{n-1,\mathcal{B}}(x_1^{n-1}). \end{cases} \quad (6.10)$$

For $c_{\mathcal{B}}(x_1^n)$ we have a similar expression. Because every state can be reached from the other one with a single transition, there is a strong relation between the weighted probability for algorithm $\mathcal{A}$ and algorithm $\mathcal{B}$: they cannot differ too much. The counts $c_{\mathcal{A}}$ and $c_{\mathcal{B}}$ cannot differ by more than one.

[1] If these counts would truly reflect the number of switches of the chosen transition sequence so far, then $c_{\mathcal{B}}$ should be initialized to 1. But it is more convenient to initialize it to 0.

It appears that we would have a free choice in case the two contributions are exactly equal. Suppose for algorithm $\mathcal{A}$ both contributions are equal. We now have to choose between $c_{\mathcal{A}}$ and $c_{\mathcal{B}} + 1$. Taking the minimum seems best, because then the number of switches will be as low as possible in the end. But because the counts cannot differ more than 1, taking $c_{\mathcal{A}}$ is identical to taking $\min(c_{\mathcal{A}}, c_{\mathcal{B}} + 1)$. Thus we simply choose $c_{\mathcal{A}}$ in this case.

The snake algorithm will not perform as well as the switching algorithm. This can be explained in two ways. At the end the snake algorithm has weighted over all possible transition sequences. The probability of the source sequence given a transition sequence, $P(x_1^n|t_1^n)$, has as weight the probability of the transition sequence $P(t_1^n)$, see equation (6.2). The switching algorithm assigns to each transition sequence the probability as computed by a KT-estimator. As a result transition sequences with fewer switches have a higher probability, and thus a higher weight. This seems a reasonable assumption. In the snake algorithm one transition sequence (the one that is being tracked, and which dictates the counts) determines the probability distribution of a transition symbol for all possible transition sequences. For most other transition sequences this probability distribution does not corresponds to its past (because it assumes either too many or too few switches) and as a result the weights assigned to these transition sequences are quite arbitrary. Thus the snake algorithm causes an unstructured probability distribution over the transition sequences, and therefore it will more often than not, result in a worse performance.

The second explanation is that the snake algorithm does not have a "switching memory". If in the switching algorithm some state (n, j) has the highest probability after n symbols, then at time $n + 1$ a much lower or much higher state can be the best one. The snake algorithm can only use more switches in the future, and only *one* extra switch per step. Thus the snake algorithm cannot "undo" previous decisions due to the lack of states (the switching memory).

The snake algorithm needs very little memory: two counters and two probabilities. Furthermore, it needs a small fixed number of operations per source symbol. Therefore, we decided to use the snake algorithm instead of the switching algorithm.

6.3 IMPLEMENTATION

The program is a direct implementation of the formulas described in the previous section, with one complication: the source symbols are not encoded in one step, but in a variable number of bits. Suppose symbol x_n has to be encoded and $c_{\mathcal{A}}(x_1^{n-1})$, $c_{\mathcal{B}}(x_1^{n-1})$, $P_w^{n-1,\mathcal{A}}(x_1^{n-1})$ and $P_w^{n-1,\mathcal{B}}(x_1^{n-1})$ are known. First the weights $w_{\mathcal{A}}(x_1^n)$ and $w_{\mathcal{B}}(x_1^n)$ will be computed.

$$w_{\mathcal{A}}(x_1^n) = (1 - P_{sw}(n-1, c_{\mathcal{A}}(x_1^{n-1}))) \cdot P_w^{n-1,\mathcal{A}}(x_1^{n-1}) + P_{sw}(n-1, c_{\mathcal{B}}(x_1^{n-1}))\, P_w^{n-1,\mathcal{B}}(x_1^{n-1}) \quad (6.11)$$

$$w_{\mathcal{B}}(x_1^n) = P_{sw}(n-1, c_{\mathcal{A}}(x_1^{n-1})) \cdot P_w^{n-1,\mathcal{A}}(x_1^{n-1}) + (1 - P_{sw}(n-1, c_{\mathcal{B}}(x_1^{n-1})))\, P_w^{n-1,\mathcal{B}}(x_1^{n-1}) \quad (6.12)$$

These weights are actually block probabilities: $w_{\mathcal{A}}(x_1^n) = P_w(t_1^n$ with an even number of trans., $x_1^{n-1})$, and similar for $w_{\mathcal{B}}(x_1^n)$. Estimate $P_{\mathcal{A}}(x_n|x_1^{n-1})$ has to be weighted with weight $w_{\mathcal{A}}(x_1^n)$ and estimate $P_{\mathcal{B}}(x_n|x_1^{n-1})$ with weight $w_{\mathcal{B}}(x_1^n)$. Next, the new counts $c_{\mathcal{A}}(x_1^n)$ and $c_{\mathcal{B}}(x_1^n)$ can be computed

with (6.10). Suppose that the forward decomposition splits symbol x_n in M bits, $z_1, \ldots, z_M$. Then

$$P_{\mathcal{A}}(x_n|x_1^{n-1}) = P_{\mathcal{A}}(z_1|x_1^{n-1})P_{\mathcal{A}}(z_2|x_1^{n-1}, z_1)\ldots P_{\mathcal{A}}(z_M|x_1^{n-1}, z_1, \ldots, z_{M-1}),$$

and similar for algorithm $\mathcal{B}$. Thus after each bit z_m, with $m = 1, \ldots, M$, the snake algorithm assigns probability,

$$P_w(x_1^{n-1}, z_1, \ldots, z_m) = w_{\mathcal{A}}(x_1^{n-1})P_{\mathcal{A}}(z_1, \ldots, z_m|x_1^{n-1}) + w_{\mathcal{B}}(x_1^{n-1})P_{\mathcal{B}}(z_1, \ldots, z_m|x_1^{n-1}) \quad (6.13)$$

to the source sequence $x_1^{n-1}, z_1, \ldots, z_m$ so far. The conditional probability of bit z_m, with $m = 1, \ldots, M$, can be computed with Bayes' rule:

$$P_w(z_m|x_1^{n-1}, z_1, \ldots, z_{m-1}) = \frac{P_w(x_1^{n-1}, z_1, \ldots, z_m)}{P_w(x_1^{n-1}, z_1, \ldots, z_{m-1})}. \quad (6.14)$$

The snake algorithm is implemented with the following steps:

- For all $n = 1, \ldots, N$ do:
  - ⋆ If $n = 1$ then
    - ▷ Initialize weights: $w_{\mathcal{A}}(x_1) = \frac{1}{2}$ and $w_{\mathcal{B}}(x_1) = \frac{1}{2}$
    - ▷ Initialize counts: $c_{\mathcal{A}}(x_1) = 0$ and $c_{\mathcal{B}}(x_1) = 0$.
  - ⋆ else
    - ▷ Compute new weights $w_{\mathcal{A}}(x_1^n)$ and $w_{\mathcal{B}}(x_1^n)$, with (6.11) and (6.12).
    - ▷ Compute new count $c_{\mathcal{A}}(x_1^n)$ with (6.10) and similar for $c_{\mathcal{B}}(x_1^n)$.
  - ⋆ Suppose $x_n = z_1, \ldots, z_M$. For all $m = 1, \ldots, M$ do:
    - ▷ Compute the new weights after m bits:

$$w_{\mathcal{A}}(x_1^{n-1}, z_1, \ldots, z_m) = w_{\mathcal{A}}(x_1^{n-1}, z_1, \ldots, z_{m-1})P_{\mathcal{A}}(z_m|x_1^{n-1}, z_1, \ldots, z_{m-1}),$$
$$w_{\mathcal{B}}(x_1^{n-1}, z_1, \ldots, z_m) = w_{\mathcal{B}}(x_1^{n-1}, z_1, \ldots, z_{m-1})P_{\mathcal{B}}(z_m|x_1^{n-1}, z_1, \ldots, z_{m-1}).$$

    - ▷ Compute the conditional probability $(*)$:

$$P_w(z_m|x_1^{n-1}, z_1, \ldots, z_{m-1}) = \frac{w_{\mathcal{A}}(x_1^n, z_1, \ldots, z_m) + w_{\mathcal{B}}(x_1^n, z_1, \ldots, z_m)}{P_w(x_1^{n-1}, z_1, \ldots, z_{m-1})}$$

    - ▷ Encode bit z_m with probability $P_w(z_m|x_1^{n-1}, z_1, \ldots, z_{m-1})$.
  - ⋆ Now $P_w^{n,\mathcal{A}}(x_1^n) = w_{\mathcal{A}}(x_1^{n-1}, z_1, \ldots, z_M)$ and $P_w^{n,\mathcal{B}}(x_1^n) = w_{\mathcal{B}}(x_1^{n-1}, z_1, \ldots, z_M)$.

These computations already have been optimized. This is reflected in the computational steps by introducing a new notation: $w_{\mathcal{A}}(x_1^{n-1}, z_1, \ldots, z_{m-1}) = w_{\mathcal{A}}(x_1^{n-1}) \cdot P_{\mathcal{A}}(z_1, \ldots, z_{m-1}|x_1^{n-1})$, and similar for $w_{\mathcal{B}}$. Now the final step in the implementation is also clear, because,

$$w_{\mathcal{A}}(x_1^{n-1}, z_1, \ldots, z_M) = w_{\mathcal{A}}(x_1^{n-1}) \cdot P_{\mathcal{A}}(z_1, \ldots, z_M|x_1^{n-1})$$

$$= w_{\mathcal{A}}(x_1^{n-1}) \cdot P_{\mathcal{A}}(x_n|x_1^{n-1}) \stackrel{\triangle}{=} P_w^{n,\mathcal{A}}(x_1^n),$$

in which the last step follows from (6.8) and (6.11). And similar for algorithm $\mathcal{B}$, of course.

Note that in the actual implementation both the probabilities of $z_m = 0$ and $z_m = 1$ are computed and normalized. This means that the actual computations differ slightly from the ones described above, and especially in the second step of the inner loop (marked with a $(*)$). In the actual implementation the conditional probability of the new bit $P_w(z_m|x_1^{N-1}, z_1^{m-1})$ is not computed by dividing the sum of the weights by $P_w(x_1^{n-1}, z_1, \ldots, z_{m-1})$, but by simply normalizing the (still undivided) conditional probabilities $P_w(Z_m = 0|x_1^{N-1}, z_1^{m-1})$ and $P_w(Z_m = 1|x_1^{N-1}, z_1^{m-1})$ instead.

Both universal source coding algorithms work with a fixed point log implementation. Their conditional probabilities are used in the inner loop of the snake algorithm (over $m = 1, \ldots, M$). The computations in the inner loop can be easily performed in the fixed-point log-domain. In the outer loop the new weights and the new counts have to be computed. The computation of the new weights requires several multiplications and additions and can be easily performed in the fixed-point log-domain too. The new counts are computed by comparing two values, but clearly these can also be compared in the fixed-point log domain, therefore also in the outer loop, and thus in the entire snake algorithm, the fixed-point log implementation will be used for the computations.

6.4 IMPROVEMENTS

The snake algorithm has an unexpected side-effect that appeared during some of the experiments. If it switches a few times during the first few symbols, the probability of a switch will be much higher than the probability of a non-switch, and as a result the snake algorithm will continue to switch after almost every symbol. Even more remarkable is that even under these circumstances the snake algorithm often gives better results than the two universal source coding algorithms it combines. We prevent this side-effect from happening by upper bounding the probability of a switch by a $\frac{1}{2}$.

For the CTW algorithm we modified the KT-estimator (see Section 4.3.2) to improve the performance and we applied count-halving. Here we use a similar approach. The estimator is controlled through the same parameter, called α_{sn} here, and we will divide the counts by two, if their sum reaches B (so, after the first B transition symbols, the counts will be halved every $B/2$ transition symbols). The snake algorithm uses a single KT-estimator, therefore one might wonder whether these modifications will influence the performance much. The answer is yes: they do influence the performance significantly. Consider the weights $w_{\mathcal{A}}$ and $w_{\mathcal{B}}$, formulas (6.11) and (6.12). These cannot differ more than by a factor which is about the probability of a switch. Next, these weights are multiplied by the corresponding estimates of the probability of the new source symbol, resulting in the weighted probabilities in the two states. Finally, these two weighted probabilities are both multiplied by the probability of a switch and the probability of a non-switch. These resulting four terms are compared in equation (6.10) to determine the new counts for algorithm $\mathcal{A}$, and similar for algorithm $\mathcal{B}$. Thus the estimated probability of the

Table 6.1: The decrease in compression speed due to the snake algorithm.

	CTW	snake		
file	sec.	sec.	%	kB/sec.
bib	11.00	11.85	7.7	9.2
book1	67.43	75.72	12.3	9.9
book2	55.16	58.59	6.2	10.2
geo	8.30	9.62	15.9	10.4
news	31.06	34.10	9.8	10.8
obj1	3.75	3.94	5.1	5.3
obj2	24.35	26.14	7.4	9.2
paper1	6.29	6.64	5.6	7.8
paper2	8.00	9.20	15.0	8.7
pic	14.00	15.56	11.1	32.2
progc	5.19	5.49	5.8	7.0
progl	7.14	8.05	12.7	8.7
progp	5.90	6.33	7.3	7.6
trans	9.70	10.49	8.1	8.7

last source symbol and the probability distribution of the last transition symbol hugely influence the updating of the counts. Therefore, the behaviour of the snake algorithm is very sensitive to modifications of the estimator, while the switching algorithm on the other hand will not be influenced much by modifying the estimator. The best set-up of the estimator for the snake algorithm is influenced by the companion algorithm (see Chapter 7).

An additional advantage of halving the counts in the snake algorithm is that there are only a limited number of possible switch probabilities. The snake algorithm uses two tables to speed up the computations. Suppose the counts are halved if their sum is B. One table represents the numerator of the KT-estimator, (6.3) and (6.4), and contains values $\log_2(b + \frac{1}{\alpha_{sn}})$, for all $b = 1, \ldots, B$, and a given α_{sn}. The second table represents the denominator of these equations ant it contains entries $\log_2(b + \frac{2}{\alpha_{sn}})$, for all $b = 1, \ldots, B$, and a given α_{sn}.

6.5 FIRST EVALUATION

Since the memory complexity of the snake algorithm, is negligible and its performance is evaluated in Chapter 7 we will only measure the impact of the snake algorithm on the compression speed here.

We measured the speed of the CTW program (in its reference set-up, with depth $D = 10$, and with the Huf-IV forward decomposition) with the snake algorithm, but without the companion algorithm, on the Calgary corpus. Table 6.1 shows the measured times, the difference in %, and the compression speed in kBytes/second (column kB/s), all averaged over ten runs, compared

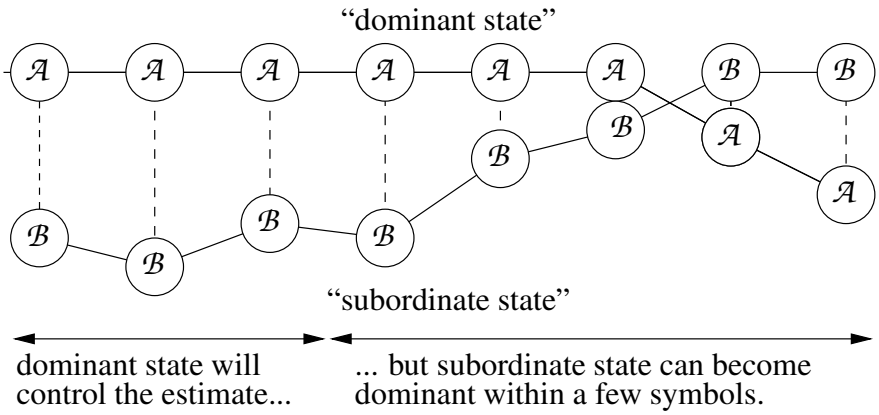


Figure 6.4: An abstract representation of the snake algorithm's behaviour.

to the compression time for this CTW algorithm alone. The CTW algorithm with the snake algorithm is on average about 10 % slower than the CTW algorithm alone.

The snake algorithm uses almost no memory: it requires a few registers to store the probabilities and the counts, and it uses two additional tables. It is also fast, it reduces the compression speed by only 10 %. Thus it can be used in the final design, if the companion algorithm fulfils the requirements too and if the snake algorithm has indeed the desired switching behaviour.

6.6 DISCUSSION

6.6.1 INTRODUCTION

There is no question about the workings of the switching algorithm. It is a full implementation, with weighting, of the switching method, and the workings and performance of this algorithm can be completely explained based on this theory. The snake algorithm is designed to work like a greedy switching algorithm, and it is supposed to try to track the best transition sequence. But does it actually work like that?

The snake algorithm has two states. We will call the state with the highest weight the dominant state (because its estimate of the probability distribution of the next symbol will dominate the overall probability distribution), and the other state the subordinate state. The behaviour of the snake algorithm can be observed by looking at the relative weight of the subordinate state compared to the dominant state. Suppose algorithm $\mathcal{A}$ is dominant with weight $w_{\mathcal{A}}$, then its relative weight is one, and the relative weight of state $\mathcal{B}$ is $\frac{w_{\mathcal{B}}}{w_{\mathcal{A}}}$. Figure 6.4 shows an example of the relative position of the weight of the subordinate state compared to that of the dominant state.

6.6.2 AN APPROXIMATION

Let us examine the properties of the interaction between the two states with a simplified case. We will also use a simplified notation: $P_{\mathcal{A}}(x_n)$ is the conditional probability for symbol x_n, and

$s_{\mathcal{A}}(x_n)$ is the transition probability for switching from state $\mathcal{A}$ to state $\mathcal{B}$ between symbol x_{n-1} and x_n, and similar for algorithm $\mathcal{B}$. The transition probabilities $s_{\mathcal{A}}$ and $s_{\mathcal{B}}$ are determined by the generalized KT-estimator (see Section 4.3.2, equation (4.9)), with count halving as soon as the sum of the counts exceeds B.

We will investigate two properties of the snake algorithm: the maximum gap between the weights of the two states and the speed with which this gap can be closed and widened. In order to get simple expressions for these two properties, we shall first derive an approximation for the new gap $G(x_{n+1})$ between the states after $n+1$ symbols, given the previous $G(x_n)$. Suppose that for a certain symbol x_n state $\mathcal{B}$ is the subordinate state, then the gap or distance is defined as $G(x_n) = \frac{w_{\mathcal{A}}(x^n)}{w_{\mathcal{B}}(x^n)}$. Next, we define a factor K, such that the transition probabilities for the new symbol x_{n+1} satisfy $s_{\mathcal{A}}(x_{n+1}) = \frac{1-s_{\mathcal{A}}(x_{n+1})}{K}$, with $K \geq 1$. This is possible because the probability of a switch $s_{\mathcal{A}}(x_{n+1}) \leq \frac{1}{2}$. Furthermore, we define the gain g as the ratio between the conditional probabilities estimated by the two algorithms: $g = \frac{P_{\mathcal{B}}(x_n)}{P_{\mathcal{A}}(x_n)}$. It is the factor that the subordinate state, say state $\mathcal{B}$, gains (or loses) compared to the dominant state. To simplify the analysis, we will assume that transition probabilities for states $\mathcal{A}$ and $\mathcal{B}$ are roughly equal and thus we use $s_{\mathcal{A}}(x_{n+1}) \approx s_{\mathcal{B}}(x_{n+1})$. With this approximation the new weights reduce to:

$$
\begin{aligned}
w_{\mathcal{A}}(x^{n+1}) &= (1 - s_{\mathcal{A}}(x_{n+1}))w_{\mathcal{A}}(x^n)P_{\mathcal{A}}(x_n) + s_{\mathcal{B}}(x_{n+1})w_{\mathcal{B}}(x^n)P_{\mathcal{B}}(x_n) \\
&\approx (1 - s_{\mathcal{A}}(x_{n+1}))w_{\mathcal{A}}(x^n)P_{\mathcal{A}}(x_n) + \frac{1 - s_{\mathcal{A}}(x_{n+1})}{K}\frac{w_{\mathcal{A}}(x^n)}{G(x_n)} \cdot g \cdot P_{\mathcal{A}}(x_n) \\
&= (1 - s_{\mathcal{A}}(x_{n+1}))w_{\mathcal{A}}(x^n)P_{\mathcal{A}}(x^n)\left(1 + \frac{g}{K \cdot G(x_n)}\right) \\
w_{\mathcal{B}}(x^{n+1}) &= s_{\mathcal{A}}(x_{n+1})w_{\mathcal{A}}(x^n)P_{\mathcal{A}}(x_n) + (1 - s_{\mathcal{B}}(x_{n+1}))w_{\mathcal{B}}(x^n)P_{\mathcal{B}}(x_n) \\
&\approx \frac{1 - s_{\mathcal{A}}(x_{n+1})}{K}w_{\mathcal{A}}(x^n)P_{\mathcal{A}}(x^n) + (1 - s_{\mathcal{A}}(x_{n+1})) \cdot \frac{w_{\mathcal{A}}(x^n)}{G(x_n)} \cdot g \cdot P_{\mathcal{A}}(x^n) \\
&= (1 - s_{\mathcal{A}}(x_{n+1}))w_{\mathcal{A}}(x^n)P_{\mathcal{A}}(x^n)\left(\frac{1}{K} + \frac{g}{G(x_n)}\right),
\end{aligned}
$$

and as a result, the new gap between the two states becomes,

$$
G(x_{n+1}) = \frac{w_{\mathcal{A}}(x^{n+1})}{w_{\mathcal{B}}(x^{n+1})} \approx \frac{K \cdot G(x_n) + g}{G(x_n) + g \cdot K}, \tag{6.15}
$$

which only depends on the previous gap $G(x_n)$, the gain g, and K.

6.6.3 THE MAXIMUM DISTANCE

The first property is the maximum distance $G^*(g)$ between the two states. The maximum gap can be computed by using equation (6.15) as a steady-state equation. If K and g are constant, then the gap $G(.)$ will converge to its maximum, independent of its current value. This maximum depends only on K and g. Assuming that K is fairly constant, we could say that to every gain g belongs a maximum gap $G^*(g)$, to which the current gap $G(.)$ would converge if this gain would be constant for a sufficient long period. So the maximum gap, the "convergence target", changes

in time, controlled by K and especially by g. An example of the computation of $G^*(g)$ can be found in Section 6.6.5.

The maximum distance between the two states is achieved if one of the algorithms estimates the probability of the new symbol as $P_{\mathcal{A} \text{ or } \mathcal{B}}(x_{n+1}) = 0$. In that case, the other state will become the dominant state, $g = 0$ and the new gap reduces to $G^*(0) = G(x_{n+1}) \approx K$, or $\log_2 K$ bits. The maximum of K is achieved if the dominant state did not have any counts yet. In that case $K = \alpha_{sn} \cdot B + 1$. In general, the factor K depends on α_{sn}, B and the current count in the dominant state. Roughly speaking, one can say that B determines the number of values that K can take, and how fast the snake algorithm can change between these levels, and α_{sn} determines the maximum distance $\alpha_{sn} \cdot B + 1$, and to a much smaller extent the distances between the other levels.

6.6.4 THE SPEED

A second important property is the speed at which the subordinate state can close the gap with the dominant state (or with which the gap can widen). Suppose that for a certain symbol x_n the subordinate state, say state $\mathcal{B}$, is at distance $G(x_n)$ from the dominant state. The new gap $G(x_{n+1})$ can be approximated by (6.15), and as a result the speed with which the gap has been closed can be expressed as,

$$\frac{G(x_n)}{G(x_{n+1})} \approx \frac{G(x_n) + g \cdot K}{K + \frac{g}{G(x_n)}} = g \cdot \frac{1}{1 + \frac{g}{KG(x_n)}} + \frac{G(x_n)}{K + \frac{g}{G(x_n)}} \approx g + \frac{G(x_n)}{K}. \tag{6.16}$$

The approximation in (6.16) holds for both closing the gap (if $g > 1$ and $G(x_n) > g$) and for widening the gap (if $g < 1$). This approximation shows that the speed is relatively constant. If the gap is at its maximum ($G(x_n) = G^*(0) = K$), then the speed with which the gap can be closed is about $g + 1$, and the speed will decrease to roughly g until it gets close to the maximum gap again.

6.6.5 A SIMULATION

Some of the behaviour of the maximum gap G^* and the speed can be shown by a very simple simulation. Suppose the probability of a switch is at its minimum: $s_{\mathcal{A}}(x_n) = \frac{1 - s_{\mathcal{A}}(x_n)}{\alpha_{sn} \cdot B + 1}$, and $s_{\mathcal{B}}(x_{n+1}) = \frac{1 + 1/\alpha_{sn}}{B + 2/\alpha_{sn}}$ (counting the switch from state $\mathcal{A}$ just before this symbol). Also, assume that the gap is at its maximum: $G(x_n) = G^*(0) = \alpha_{sn} \cdot B + 1$, e.g. caused by a zero estimate from one of the two algorithms. We will use $B = 128$ and $\alpha_{sn} = 2$ (the KT-estimator). If algorithm $\mathcal{B}$ has a constant gain of $g = 2$ on all source symbols from now on, how do the relative weights of the two states behave? In Figure 6.5 the weight of state $\mathcal{A}$ has been set to one (zero bits), and the weight of state $\mathcal{B}$ is computed relatively to this state. The gap is at first $G_1 = G^*(0) = -\log_2(\alpha_{sn} B + 1) \approx -8.0$ bits. The initial speed is almost $g + 1 = 3$ as indicated by the auxiliary line $\log_2(\frac{3^x}{257})$. It then drops to about a factor of $g = 2$, as indicated by the auxiliary line $\log_2(\frac{2^x}{257})$ and the tangent (denoted by $\sim \log_2(2^x)$). Only when the gap is nearing its maximum value again, the speed drops. The gap is now $G_2 = G^*(2) \approx 5.4$ bits. This is lower

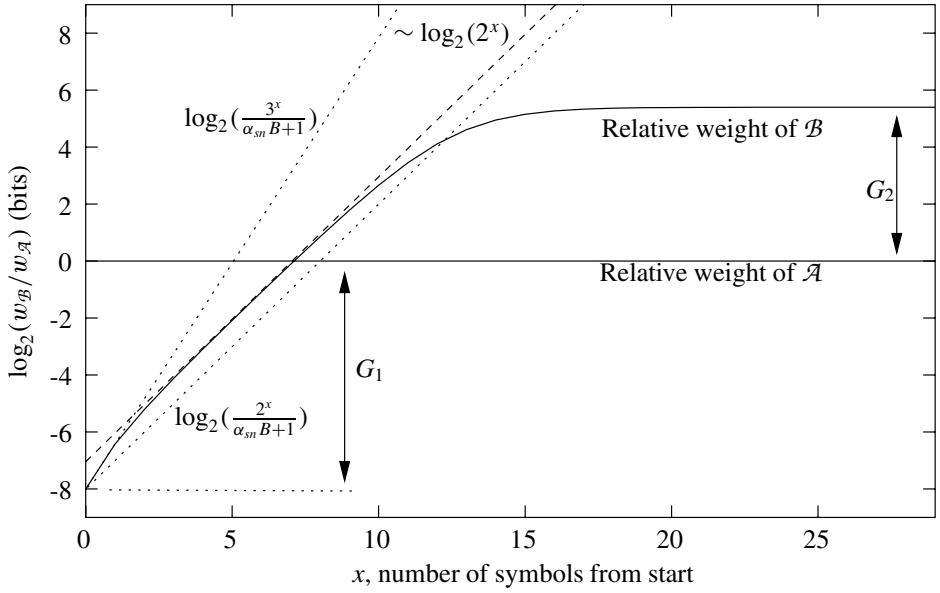


Figure 6.5: A simplified simulation of the relative weights.

than the maximum gap with 1 count $((B-1+\frac{1}{\alpha_{sn}})/(1+\frac{1}{\alpha_{sn}}) = 85 = 6.4$ bits, because the relative weights are now in equilibrium, and the weight of state $\mathcal{A}$ not only gets a contribution from state $\mathcal{B}$, but also a significant contribution from itself. Our approximation of the gap, equation (6.15), with the correct values, $g = 2$ and $K = 85$, results indeed in the equilibrium distance, $G_2 = 42.5 = 5.4$ bits.

6.6.6 CONJECTURE

In summary, the snake algorithm has two states, and the interaction between the two states can be described by two properties: the maximum distance between the states, and the speed with which one state can move relatively to the other one. The snake algorithm appears to keep the speed more or less fixed to g, but it adaptively changes the maximum distance between the two states, based on the current gain g, and on the factor K. This important factor K is controlled by the count (which locally counts the number of switches), and by α_{sn} and B. Thus by modifying our two parameters, we modify the adaptive behaviour of the snake algorithm: we directly influence the value and the dynamic behaviour of the maximum distance between the two states.

So, does the snake algorithm work like a greedy switching algorithm, and does it try to track the best transition sequence? Our conjecture is that the mechanisms of the snake algorithm describe in a complicated way a heuristic algorithm, that on one hand prevents the subordinate state from floating too far away from the dominant state (but still far enough to minimize its influence on the estimate), while on the other hand it allows the subordinate to overtake the dominant state within a few symbols. This conjecture is not only based on the discussion in this section, but it is also justified by the optimal choice for B and α_{sn} (see Chapter 7). A very small

B is chosen, while if the snake algorithm would indeed resemble the switching algorithm more, a much higher value for B might be preferable (to get a more accurate estimate of the actual switching probability). Maybe, with this new point of view in mind, a more efficient and better performing heuristic algorithm can be found?

Does this mean that the snake algorithm is useless, or at least very inefficient? On the contrary, the snake algorithm has a pleasantly low complexity and (based on its compression performance) it achieves an excellent switching performance. Experiments show that extremely simplified heuristic algorithms that have only one fixed maximum distance and that use a fixed multiplication factor to compute the speed from g, do not perform as well as the snake algorithm. It appears that some form of adaptivity to the local circumstances (the local probability of a switch, the local gain, etc.) is necessary to achieve a reasonable performance.

7

SUBSYSTEM 5: THE COMPANION ALGORITHM

This chapter discusses the implementation of a companion algorithm, a data compression algorithm that should run in parallel to the CTW algorithm. The companion algorithm should be complementary to CTW, such that it cancels some of the weaknesses of CTW. We choose a PPM algorithm with a high maximum depth, because it will result in an excellent performance on long repetitive strings. In the end we designed a new implementation of PPMA. In this new implementation PPMA has been decomposed such that it can now both be plugged into the snake algorithm without any modification, and at the same time it can share the data structures of the CTW algorithm. The memory complexity of the resulting design has increased by only 2 MBytes, but the compression speed has decreased to only 6.6 kBytes/second. The new design meets the strong performance requirement.

7.1 DESCRIPTION

THIS chapter is concerned with the companion algorithm. The companion algorithm is a universal source coding algorithm, thus it should estimate the probability distribution of each source symbol. It will only be used as the algorithm that is running in parallel to CTW, therefore it does not need to have a very good overall performance, but it must only achieve a good performance on those parts of the sequence that are not compressed well by CTW. Because the companion algorithm will only be used in combination with the snake algorithm, the requirements of these two subsystems cannot be separated easily. The final implementation of these two subsystems has to fulfil the following requirements.

- The companion algorithm, together with the snake algorithm, should improve the perfor-

mance of the final design on non-text files, such that the complete system fulfils the strong performance requirement.

- The companion algorithm should result in only a small increase in memory and computational complexity.

First a PPM algorithm, called PPMA, will be discussed. This algorithm originally processes entire source symbols, therefore we have to modify it such that it can process bits and such that it can use the information already stored in the context trees. Next, we present a small additional data structure, such that the PPMA algorithm can use much deeper contexts than the maximum depth of the context trees, in order to improve its performance as companion algorithm. Finally, its implementation will be discussed and the design will be evaluated.

7.2 BASIC ALGORITHM

7.2.1 INTRODUCTION

The companion algorithm should be complementary to the CTW algorithm. One of the weaker points of the CTW algorithm is its performance on long repetitive strings in the source sequence. Such a string has to occur several times before the counters in the corresponding nodes and leaves in the context trees are biased enough to have a dominant effect on the weighted probability for those contexts. A Lempel-Ziv algorithm on the other hand can efficiently compress such a string, the second time it occurs. If we choose a Lempel-Ziv-like algorithm as companion algorithm and combine it with the CTW algorithm, then our final design would incorporate this desirable behaviour too, possibly improving the performance.

Lempel-Ziv-like algorithms are either based on the Lempel-Ziv'77 algorithm [72], or on the Lempel-Ziv'78 algorithm [73]. We will not discuss these algorithms in detail. Roughly said, Lempel-Ziv algorithms encode a string of new source symbols simultaneously, by replacing the entire string with a reference to a previous occurrence of this string in the source sequence. Lempel-Ziv'77 algorithms can refer to any string of consecutive symbols of any length observed in the past, and therefore a long repetitive string can be completely matched and replaced the second time it occurs. Lempel-Ziv'78 algorithms use a more restricted dictionary of previously observed strings and match the new source symbols with strings from this dictionary. After coding a string, only this single string, extended by the first unmatched source symbol, is added to the dictionary. Therefore, the dictionary grows slowly, and also the maximum length of a repetitive string that can be replaced at once grows slowly. For our purposes, clearly the behaviour of Lempel-Ziv'77-like algorithms is preferred. Lempel-Ziv algorithms cannot be used as companion algorithm directly, because originally these algorithms do not strictly split modelling and coding. Although, these algorithms can be split into a modelling part and a coding part (for the Lempel-Ziv'78 algorithm see e.g. [22]), this is not necessary in our case, because our companion algorithm does not have to mimic the behaviour of a Lempel-Ziv algorithm exactly, it only has to have a Lempel-Ziv'77-like behaviour on long repetitive strings.

Here, we will focus on the family of Prediction by Partial Matching (PPM) algorithms, which are easier to integrate in our design, because they strictly split modelling and coding. The first members of this family were published in 1984 [13], and from then on there has been a tremendous amount of research on PPM algorithms, resulting in a wide range of variations: PPMA and PPMB [7, 13], PPMC [7, 27], PPMD [18], PPM$*$ [12], state selection [10], and PPMZ [9], among others. PPM algorithms estimate the probability distribution for the next context symbol for different lengths of the context, and mix these estimates to obtain one probability distribution. Most PPM algorithms examine contexts up to a fixed length, say D_{PPM}, but e.g. PPM$*$ uses unbounded context lengths. In the original paper [12] it is already mentioned that PPM$*$ resembles Lempel-Ziv'78-like algorithms. In [57] it is shown that there is a close relationship between Lempel-Ziv'77 algorithms and PPM$*$ and that actually PPM$*$ gives a superior performance when it is being used as a companion algorithm. Unfortunately, also PPM$*$ cannot be implemented easily in our final design, but other PPM algorithms with a high maximum depth will have a similar behaviour on long matches and can be implemented easily. Here we chose to use PPMA.

A PPM algorithm estimates the probability distribution of an entire source symbol. The estimation process consists basically of two parts. First, for each length of the context up to D_{PPM}, a "memoryless" estimate is computed based on the counts of the symbols following that context. Next, all these estimates of the probability distribution of the next symbol, plus a uniform distribution are mixed in one final probability distribution. The following section will describe the memoryless estimator for PPMA, and the section after that discusses the mixing algorithm.

7.2.2 THE MEMORYLESS ESTIMATOR FOR PPMA

For every context that occurred, and for every possible depth, a PPM algorithm counts how often each source symbol followed that context in the source sequence so far. These counts are used by the memoryless estimator of the PPM algorithm to compute the estimated probability distribution of the next source symbol. The different PPM algorithms use different ways to estimate this probability distribution. They have in common that they assign a positive probability to only those source symbols that followed that context in the source sequence in the past. PPMA estimates at depth d the probability that symbol x_n follows the context s by

$$P_e^{s,d}(X_n = x|x_1^{n-1}) = \frac{\#x_s}{C_s} \tag{7.1}$$

for all depths $d \triangleq l(s) = 0, 1, \ldots, D_{PPM}$ and with $s = x_{n-d} \ldots x_{n-1}$. Here $\#x_s$ is the number of times that symbol x followed context s in the source sequence so far, and C_s is the number of times that this context s occurred. Note that similar to Chapter 4, we use a simplified notation, because both $\#x_s$ and C_s depend on the sequence x_1^{n-1}. Source symbols that did not occur at all in the source sequence yet, will not get a positive probability estimate at any context length from $d = 0$ to $d = D_{PPM}$. Therefore, an additional depth $d = -1$ is introduced, at which each possible source symbol x is estimated by the uniform estimator $P_e^{-,-1}(x) \triangleq \frac{1}{|\mathcal{X}|}$, in which $|\mathcal{X}|$ is the size of the source symbol alphabet.

7.2.3 COMBINING THE MEMORYLESS ESTIMATES IN PPMA

These estimates from the different context lengths have to be mixed together. This is handled by an *escape mechanism*. The underlying idea of PPM is that source symbols should be estimated at the longest possible context length[1]. Therefore, these algorithms typically investigate the longest context length first. But it is very well possible that the next source symbol did not follow this context yet. In that case the algorithm needs to escape to a context one symbol shorter, and check whether the next source symbol has already followed that context. The PPM algorithms implement this mechanism by estimating for each context length the probability that the next source symbol is "new" for this context length, and that they thus have to escape to a shorter context. The way in which this escape probability is computed differs for the various PPM algorithms. PPMA uses escape method A, which defines the escape probability for context s at a certain depth d as:

$$P_{\text{esc}}^{s,d}(x_1^{n-1}) = \frac{1}{C_s + 1}, \tag{7.2}$$

for all contexts s with length $d = 0, \ldots, D_{PPM}$, and $P_{\text{esc}}^{-,-1} = 0$. This particular choice is based on the simple assumption that the more often context s has occurred so far, the less probable it is that a new symbol will follow this context now. It is obvious that this is a rather disputable choice. For example, even if all possible source symbols already followed context s, then still the escape probability would be positive. Unfortunately, our choice for the computation of the escape probability is limited, because we want to use as much of the existing data structures as possible. For example, PPMC and PPM$*$ use an escape probability based on the number of different symbols following each context. But such "global" information cannot be easily retrieved from the existing, decomposed, context trees for the CTW algorithm, which makes it impossible to easily implement these PPM algorithms in our final design, although these algorithms might result in a superior performance as indicated in [57].

A PPM algorithm reaches depth d^* after escaping $D_{PPM} - d^*$ times from a longer context. Thus the weight associated with the estimate of context s at depth d is given by:

$$w_{s,d^*}(x_1^{n-1}) = (1 - P_{\text{esc}}^{s,d^*}(x_1^{n-1})) \prod_{\substack{d=d^*+1 \text{ with} \\ s=x_{n-d}\ldots x_{n-1}}}^{D_{PPM}} P_{\text{esc}}^{s,d}(x_1^{n-1}), \tag{7.3}$$

for all depths $d = -1, \ldots, D_{PPM}$. Now the estimates at the different depths can be mixed together,

$$P_{\text{PPM}}(x_n|x_1^{n-1}) = \sum_{\substack{d=-1 \text{ with} \\ s=x_{n-d}\ldots x_{n-1}}}^{D_{PPM}} w_{s,d}(x_1^{n-1})\, P_e^{s,d}(x_n|x_1^{n-1}). \tag{7.4}$$

Interestingly, this mixing technique, called blending in [7], does not completely match the idea behind PPM. PPM escapes to a shorter context to code source symbols that did not follow the longer contexts yet. But the PPM algorithm described here, gives a positive probability to a

[1] Some PPM variations, like "state selection" and PPMZ, do not start at the longest possible context length, but use other criteria instead to select for each symbol the starting context length.

symbol for each depth at which it occurred so far, and not only for the longest context length. This can be resolved by *count exclusion*: once a symbol occurs at a certain depth, its counts will be excluded from the computations at the lower depths. We decided not to use the exclusion technique because it cannot be implemented easily in our final design. If necessary we will try to change the behaviour of our PPM algorithm by changing the computation of the escape probability, and thus by changing the weights.

For us, the most important property of the PPM algorithm is that longer contexts get in general higher weights than shorter contexts. This is an inherent feature of all PPM algorithms: the symbols following the longest matching context will dominate the final probability distribution. The model grows as fast as possible. This realizes the desired Lempel-Ziv-like behaviour of the PPM algorithm: in the PPM algorithm longer contexts, and thus repetitive strings, will influence the probability immediately. The CTW algorithm on the other hand grows the model slowly: a node of the context tree can only become part of the model if it has proven on the source sequence so far that its inclusion in the current model improves the performance. In this perspective the behaviour of a PPM algorithm is indeed complementary to the CTW algorithm.

Discussion. At first sight, PPM and CTW might not differ much. They both estimate the conditional probability for each context length, and they both weight the probabilities. But, any closer investigation will show that these two algorithms are completely different. CTW sets a-priori fixed weights for every possible tree source. It then weights over all possible tree sources $\mathcal{S}$: $P_w(x_1^N) = \sum_{S \in \mathcal{S}} w_S P_e(x_1^N|S)$. Because it weights the block probabilities of the entire source sequence, the behaviour of the CTW algorithm can be easily analysed. At each moment, the CTW algorithm loses not more than $\log_2 w_{S^*}$ bits, compared to an algorithm that knows the optimal model S^*. The PPM algorithms on the other hand weight over the *conditional* probabilities at each depth. This might still not seem too different from CTW, because in Section 4.4.1, a factor β was introduced, which allowed us to reformulate the CTW algorithm as a series of weightings of conditional probabilities too. But the main difference between this rewritten CTW algorithm and PPM is the computation of the weights. The CTW algorithm uses in each node a factor β as weight which is an *exact* representation of the performance of that node on the source sequence so far (see equation (4.14)). While the PPM algorithms on the other hand use for a certain context an almost arbitrary weight based on an experimentally determined escape probability. Consequently, in PPM algorithms there is no direct link between the weight of a node and its past performance. This makes the analysis of PPM algorithms, and their asymptotic behaviour, extremely difficult. As a result many publications on PPM simply show a new variation and experimentally determine its (often very good) performance. There are some exceptions, e.g. [10, 11] give an analysis of PPM, and most noticeably is the excellent work in [1], in which a very comprehensive theoretical analysis of PPM is given.

Example, from [7]: Suppose the source sequence generated by a 5-ary source starts with cacbcaabca and the PPM algorithm has maximum depth $D_{PPM} = 4$. Table 7.1 shows for each context depth the symbol counts so far, and the resulting estimated probability $P_e^{s,d}$ at that context depth. The last two columns show the escape probabilities and the resulting weights for each depth. The bottom row shows the weighted probabilities of all source symbols.

Table 7.1: The weights, the various probabilities and the counts of methods A, after the sequence: "cacbcaabca".

	Context	Counts	Estimates – $P_e^{s,d}(.\|.)$ $\#._s$										Method A	
d	s	C_s	a		b		c		d		e		$P_{\text{esc}}^{s,d}$	$w_{s,d}$
4	abca	0	–	0	–	0	–	0	–	0	–	0	1	0
3	bca	1	1	1	0	0	0	0	0	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$
2	ca	2	$\frac{1}{2}$	1	0	0	$\frac{1}{2}$	1	0	0	0	0	$\frac{1}{3}$	$\frac{1}{3}$
1	a	3	$\frac{1}{3}$	1	$\frac{1}{3}$	1	$\frac{1}{3}$	1	0	0	0	0	$\frac{1}{4}$	$\frac{1}{8}$
0		10	$\frac{4}{10}$	4	$\frac{2}{10}$	2	$\frac{4}{10}$	4	0	0	0	0	$\frac{1}{11}$	$\frac{5}{132}$
-1	–	–	$\frac{1}{5}$	–	$\frac{1}{5}$	–	$\frac{1}{5}$	–	$\frac{1}{5}$	–	$\frac{1}{5}$	–	0	$\frac{1}{264}$
mixed probabilities			$\frac{956}{1320}$		$\frac{66}{1320}$		$\frac{296}{1320}$		$\frac{1}{1320}$		$\frac{1}{1320}$			

7.3 INTEGRATING PPMA IN THE FINAL DESIGN

7.3.1 RESTRICTIONS FOR OUR PPMA IMPLEMENTATION

The PPM algorithm as described in the previous section cannot be integrated directly into our final design; first two implementation issues have to be resolved. First of all, the PPM algorithm is not allowed to use a lot of extra memory for its own data structure, while it requires information that is not readily available in the context trees, e.g., the counts of the source symbols that followed every context so far. Secondly, the snake algorithm has to process either entire source symbols or bits, and CTW processes bits and the PPM algorithm entire source symbols. This second issue can be easily fixed by computing the PPM estimates for the individual bits from its estimate of the probability distribution over the entire source alphabet. The alternative solution for the second issue, computing the CTW estimate over the entire source alphabet, would require an estimate from every context tree and will increase the computational complexity enormously. But, instead of trying to patch the interface between the algorithms, we decided to design a new decomposed implementation of the PPMA algorithm, that solves both problems: it will estimate the probability distribution of the individual bits as described by the forward decomposition, based only on the information already available in the context trees.

7.3.2 DECOMPOSING PPMA

We will first explain the new decomposed PPMA implementation from a general point of view before we illustrate the process by means of a simple example in the next section.

Assume that symbol x_n with context s has to be coded. Just as the regular PPMA algorithm, the decomposed PPMA implementation first computes the weights $w_{-,-1}(x_1^{n-1}), \ldots, w_{s,D_{PPM}}(x_1^{n-1})$ with equations (7.2) and (7.3).

Suppose symbol x_n is decomposed in some M bits $x_n = z_1, \ldots, z_M$. The decomposed PPMA algorithm has to estimate the probability of each bit z_m with $m = 1, \ldots, M$. For each bit z_m it

first estimates the block probability of the first m bits,

$$P_{\text{PPM}}(z_1^m|x_1^{n-1}) = \sum_{\substack{d=-1 \text{ with} \\ s=x_{n-d}\ldots x_{n-1}}}^{D_{PPM}} w_{s,d}(x_1^{n-1})\, P_e^{s,d}(z_1^m|x_1^{n-1}), \tag{7.5}$$

before it applies Bayes' rule to compute the conditional probability:

$$P_{\text{PPM}}(z_m|z_1^{m-1}, x_1^{n-1}) = \frac{P_{\text{PPM}}(z_1^m|x_1^{n-1})}{P_{\text{PPM}}(z_1^{m-1}|x_1^{n-1})}, \tag{7.6}$$

with $P_{\text{PPM}}(\emptyset|x_1^{n-1}) = 1$. Clearly, $\prod_{m=1}^{M} P_{\text{PPM}}(z_m|z_1^{m-1}, x_1^{n-1}) = P_{\text{PPM}}(z_1^M|x_1^{n-1})$, thus the decomposed PPMA algorithm achieves the same probability distribution as the regular PPMA algorithm. Furthermore, since

$$P_{\text{PPM}}(Z_m = 0, z_1^{m-1}|x_1^{n-1}) + P_{\text{PPM}}(Z_m = 1, z_1^{m-1}|x_1^{n-1}) = P_{\text{PPM}}(z_1^{m-1}|x_1^{n-1}),$$

holds, (7.6) is a good coding distribution and can be used in the snake algorithm or by an arithmetic coder.

This straightforward decomposition has one remaining problem. To compute (7.5) in every node on the context path, the estimated block probability $P_e^{s,d}(z_1^m|x_1^{n-1})$ over all bits so far has to be computed, while the context tree for bit z_m only contains the (scaled) counts for this single bit. This is solved by computing these estimated probabilities as:

$$P_e^{s,d}(z_1^m|x_1^{n-1}) = P_e^{s,d}(z_m|z_1^{m-1}, x_1^{n-1}) \cdot P_e^{s,d}(z_1^{m-1}|x_1^{n-1}). \tag{7.7}$$

The conditional probability $P_e^{s,d}(z_m|z_1^{m-1}, x_1^{n-1})$ can be easily computed from the counts in node s of the context tree for bit z_m. Because all bits of one source symbol have the same context s, the second term, $P_e^{s,d}(z_1^{m-1}|x_1^{n-1})$ is the estimated probability computed at the same depth in the context tree used for the previous bit z_{m-1}. Therefore, to allow for an efficient computation of (7.7), and consequently of (7.5), at each depth in each context tree this block probability should be stored for the next bit. We call this block probability the correction factor δ, $\delta_{s,d}(z_1^{m-1}) \triangleq P_e^{s,d}(z_1^{m-1}|x_1^{n-1})$. The need for such a correction factor that is being carried from one context tree to the next shows one important difference between the CTW algorithm and the PPMA algorithm. Because the PPMA algorithm processes entire source symbols, the estimates in the context trees for the consecutive bits of one source symbol are not independent.

7.3.3 SOME IMPLEMENTATION CONSIDERATIONS

Let us investigate the implementation of the decomposed PPMA algorithm closer with a simple example. Consider a 4-ary source and the decomposition in Figure 7.1 and assume that $x_n = a$. For reference, the regular PPMA algorithm first computes the weights $w_{-,-1}, \ldots, w_{s,D_{PPM}}$ with equations (7.2) and (7.3). Next, it estimates the probability of symbol "a" by

$$P_{\text{PPM}}(X_n = a|x_1^{n-1}) = \sum_{\substack{d=0 \text{ with} \\ s=x_{n-d}\ldots x_{n-1}}}^{D_{PPM}} w_{s,d}(x_1^{n-1}) \frac{\#a_s}{\#a_s + \#b_s + \#c_s + \#d_s} + w_{-,-1}(x_1^{n-1})\frac{1}{4}. \tag{7.8}$$

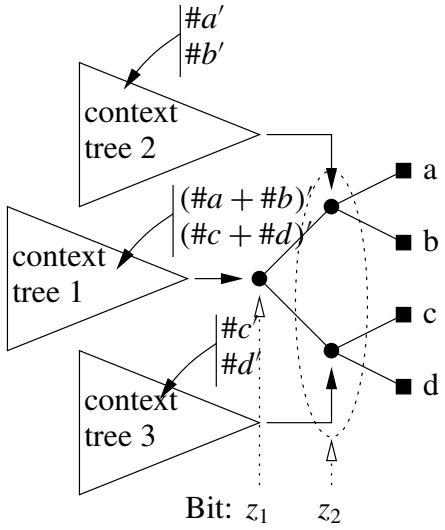


Figure 7.1: A simple example.

Our decomposed PPMA implementation also computes first the weights $w_{-,-1}, \ldots, w_{s,D_{PPM}}$ with equations (7.2) and (7.3). The weights are determined by the escape probabilities, which depend on the number of symbols following each context. These counts are obtained from the context tree corresponding to the root-node of the forward decomposition. This brings out an other problem, the context trees do not contain the precise counts $\#a_s$, but only scaled versions, $\#a'_s$, of these counts. For the computation of the weights, it doesn't really matter whether these counts were scaled in the past, because counts will only be scaled if there sum is large, and even after scaling their sum will still be large, and as a result the escape probability $P_{\text{esc}}^{s,d}$ will be so small that shorter contexts will get a negligible weight anyway.

Next, our decomposed PPMA implementation should estimate the probability of the first bit, z_1, of the decomposed symbol, the probability that $X_n = a$ or b, with

$$P_{\text{PPM}}(X_n = a \text{ or } b | x_1^{n-1}) = \sum_{\substack{d=0 \text{ with} \\ s=x_{n-d}\ldots x_{n-1}}}^{D_{PPM}} w_{s,d}(x_1^{n-1}) \frac{\#a_s + \#b_s}{\#a_s + \#b_s + \#c_s + \#d_s} + w_{-,-1}(x_1^{n-1}) \frac{1}{2}. \quad (7.9)$$

The computation of the probability distribution for the first bit (7.9) uses the quotient $\frac{\#a_s+\#b_s}{\#a_s+\#b_s+\#c_s+\#d_s}$. This quotient can be approximated with the (scaled) counts, available in context tree 1,

$$\frac{\#a_s + \#b_s}{\#a_s + \#b_s + \#c_s + \#d_s} \approx \frac{(\#a_s + \#b_s)'}{(\#a_s + \#b_s)' + (\#c_s + \#d_s)'}.$$

The ratio between the counts will not be significantly affected by the scaling operation, therefore, the penalty for using the counts from the context tree will be small compared to the other approximations.

Finally, our decomposed PPMA algorithm has to estimate the probability of the second bit

z_2, the probability that the new symbol x_n is an "a", given that it is an "a" or a "b". Note that

$$P_{\text{PPM}}(X_n = a|x_1^{n-1},\, X_n = a \text{ or } b) \neq \sum_{\substack{d=0 \text{ with} \\ s=x_{n-d}\ldots x_{n-1}}}^{D_{PPM}} w_{s,d}(x_1^{n-1}) \frac{\#a_s}{\#a_s + \#b_s} + w_{-,-1}(x_1^{n-1}) \frac{1}{2}. \quad (7.10)$$

As explained above, our PPMA algorithm first computes $P_{\text{PPM}}(X_n = a|x_1^{n-1})$ with (7.8) and then uses Bayes' rule to compute the probability distribution of the second bit:

$$P_{\text{PPM}}(X_n = a|x_1^{n-1},\, X_n = a \text{ or } b) = \frac{P_{\text{PPM}}(X_n = a|x_1^{n-1})}{P_{\text{PPM}}(X_n = a \text{ or } b|x_1^{n-1})}. \quad (7.11)$$

In order to compute $P_{\text{PPM}}(X_n = a|x_1^{n-1})$ with (7.8), the PPM algorithm needs to compute the quotient $\frac{\#a_s}{\#a_s+\#b_s+\#c_s+\#d_s}$. But this quotient cannot be computed with the information in context tree 2 alone. But it can be approximated by,

$$\begin{aligned} \frac{\#a_s}{\#a_s + \#b_s + \#c_s + \#d_s} &= \frac{\#a_s}{\#a_s + \#b_s} \cdot \frac{\#a_s + \#b_s}{\#a_s + \#b_s + \#c_s + \#d_s} \\ &\approx \frac{\#a'_s}{\#a'_s + \#b'_s} \cdot \frac{(\#a_s + \#b_s)'}{(\#a_s + \#b_s)' + (\#c_s + \#d_s)'}. \end{aligned} \quad (7.12)$$

Here, the first term is the conditional probability $P_e^{s,d}(z_2|z_1, x_1^{n-1}) = \frac{\#a_s}{\#a_s+\#b_s} \approx \frac{\#a'_s}{\#a'_s+\#b'_s}$, which can be approximated by the counts in context tree 2. The second term is the correction factor $\delta_{s,d}(z_1) = P_e^{s,d}(z_1|x_1^{n-1}) \approx \frac{(\#a_s+\#b_s)'}{(\#a_s+\#b_s)'+(\#c_s+\#d_s)'}$ as it has been stored while coding the previous bit z_1 with context tree 1.

7.3.4 EXTENDING THE RANGE

Earlier experiments [56, 57] showed that the depth of the PPM algorithm is very important. Although most PPM algorithms perform best with a maximum depth of around 5 symbols, PPM algorithms used as companion algorithm should have a much higher depth, since for our purpose, we are more interested in their Lempel-Ziv-like behaviour, than in their overall performance. The algorithm described in the previous section implements a decomposed PPMA algorithm that uses only the counts in the context trees. But this limits the maximum PPM depth to the maximum depth of the context trees. It might be worthwhile to investigate a strategy which allows the PPM algorithm to use longer contexts.

Chapter 4 discussed the implementation of the CTW algorithm. Two possible techniques to extend unique paths were presented. First, up to three context symbols can be stored in every leaf. Secondly, a (limited) buffer can be used to store (a part of) the file so far and each leaf has a pointer to the last occurrence of its context in the file. This file buffer can also be used by the PPM algorithm to find the, almost arbitrary long, context of each recently created (or visited) leaf. This is one of the main reasons why the limited file buffer is used in the CTW implementation.

The file buffer alone, together with the index from the leaf in the context tree is not sufficient for our PPM algorithm to extend its contexts, because it needs to know *all* occurrences of the

extended contexts, in order to estimate the next source symbol accurately, instead of only the last one. Therefore, the PPM implementation with the extended maximum depth needs an additional buffer, the pointer buffer, of the same length as the file buffer. This pointer buffer is used to form a linked list of all contexts that have the same first D context symbols. From a leaf in the context tree the program can find the corresponding index in the file buffer. At the same index in the pointer buffer the linked list starts, and it will find a pointer to the next context with the same prefix. And so on. From all these contexts it can (partly) reconstruct the counts on the context path between depths $D+1$ and D_{PPM}, and in this way effectively increase the maximum depth of the PPMA algorithm.

7.4 IMPLEMENTATION

For a correct operation of the decomposed PPMA algorithm, it has to transfer information between the context trees of consecutive bits by means of a correction factor $\delta_{s,d}$. For the first bit, these correction factors are initialized to:

$$\delta_{s,d}(\emptyset) = 1, \quad \text{for all } -1 \leq d \leq D_{PPM}, \tag{7.13}$$

and after coding bit z_m the correction factors have to be updated:

$$\delta_{s,d}(z_1^m) \stackrel{\triangle}{=} P_e^{s,d}(z_1^m | x_1^{n-1}), \quad \text{for all } -1 \leq d \leq D_{PPM}. \tag{7.14}$$

The complete PPMA algorithm can now be described with the following steps:

- For all x_n, with $n = 1, \ldots, N$, do:
  - ⋆ Initialize correction factors (7.13). Select the context tree corresponding to the root node of the forward decomposition.
  - ⋆ Compute the new weights:
    - ▷ (1) Use this context tree and the linked list corresponding to this context to create a table with for each context length $d = 0, \ldots, D_{PPM}$, the number of occurrences of that context.
    - ▷ Compute the new weights $w_{s,d}(x_1^n)$, for all $d = -1, 0, \ldots, D_{PPM}$, and $s = x_{n-d}, \ldots, x_{n-1}$, with (7.2) and (7.3).
  - ⋆ Code x_n. Suppose $x_n = z_1, \ldots, z_M$. For all z_m, with $m = 1, \ldots, M$ do:
    - ▷ (2) Use the counts from the currently selected context tree (and the records from the linked list corresponding to the context) to create a table with for each depth $d = 0, \ldots, D_{PPM}$ the number of occurrences of a zero and a one for this bit.
    - ▷ For all depths $d' = D_{PPM}, \ldots, -1$, with $s = x_{n-d'}, \ldots, x_{n-1}$, do:

◇ Compute the corrected ratio of the counts at this depth:

$$P_e^{s,d'}(z_m = 0, z_1^{m-1}|x_1^{n-1}) = \begin{cases} \dfrac{a_s}{a_s + b_s} \cdot \delta_{s,d'}(z_1^{m-1}), & \text{if } d' \geq 0 \\ \dfrac{1}{2} \cdot \delta_{-,-1}(z_1^{m-1}), & \text{if } d' = -1, \end{cases}$$

and similar for $z_m = 1$.

◇ Compute $P_{\mathsf{PPM}}^{s,d'}(z_1^m|x_1^{n-1})$, the weighted sum so far,

$$P_{\mathsf{PPM}}^{s,d'}(z_1^m|x_1^{n-1}) \stackrel{\triangle}{=} \sum_{\substack{d=d' \text{ with} \\ s=x_{n-d}\ldots x_{n-1}}}^{D_{PPM}} w_{s,d} P_e^{s,d}(z_1^m|x_1^{n-1}) = w_{d'} P_e^{s,d'}(z_m, z_1^{m-1}|x_1^{n-1}) + P_{\mathsf{PPM}}^{s,d'+1}(z_1^m|x_1^{n-1}),$$

for both $z_m = 0$ and $z_m = 1$.

▷ Compute the conditional probability of z_m, by normalizing the weighted probabilities (note that $P_{\mathsf{PPM}}(z_1^m|x_1^{n-1}) \stackrel{\triangle}{=} P_{\mathsf{PPM}}^{-,-1}(z_1^m|x_1^{n-1})$):

$$P_{\mathsf{PPM}}(z_m = 0|x_1^{n-1}, z_1^{m-1}) = \frac{P_{\mathsf{PPM}}(z_1^{m-1}, z_m = 0|x_1^{n-1})}{P_{\mathsf{PPM}}(z_1^{m-1}, z_m = 0|x_1^{n-1}) + P_{\mathsf{PPM}}(z_1^{m-1}, z_m = 1|x_1^{n-1})},$$

and similar for $z_m = 1$.

▷ Use the arithmetic coder to encode or decode z_m with $P_{\mathsf{PPM}}(z_m|x_1^{n-1}, z_1^{m-1})$. If the PPMA algorithm is embedded in the final design, it will code this bit through the snake algorithm.

▷ Update the correction factors $\delta_{s,d}(z_1^m)$ for all $d = -1, \ldots, D_{PPM}$ and $s = x_{n-d}, \ldots, x_{n-1}$, with (7.14).

▷ Increment a_s if $z_m = 0$ or b_s if $z_m = 1$, in the context tree for all $d = 0, \ldots, D_{PPM}$. Note that if the PPMA algorithm is embedded in the final design, the CTW algorithm will already take care of this.

▷ Go to the correct child in the forward decomposition and select its context tree.

⋆ Update the file buffer, the pointer buffer.

This is a simplified outline of the actual implementation. Especially steps marked (1) and (2) are more complicated. Because the amount of memory is fixed, it is possible that for one bit of a decomposed symbol the context path ends at a certain depth in the context tree (because its child node could not be created), while the context path in the context tree corresponding to the next bit goes deeper. Such a situation has to be prevented, because there are no correction factors, and possibly no weights, for the longer contexts in the second tree. Therefore, the implementation has to take care that for the consecutive bits of a symbol, the maximum depth used by the PPM algorithm does not increase.

The computations consist of multiplications, divisions and additions. These can be easily performed with the fixed-point log-implementation. Furthermore, we use three tables to increase the speed. The first table gives the fixed-point log-representation of the integer numbers up to 512. This table is used to compute the numerator and the denominator in the computation of the estimated probability (7.1). The second table gives the value of the escape probability (7.2) for every possible number of symbols in a state. The third table gives the fixed-point log-representation of $1-p$ for every p and it is used for the computation of the weights, (7.3). This table has only a 2183 entries (see also section 4.4.1), because of the limited accuracy.

7.5 IMPROVEMENTS

The algorithm implemented for the final design is a PPMA algorithm with blending. PPM algorithms have been improved in many different ways, and many of these variations can be decomposed in a similar way, but for the short contexts we are restricted in our choice by the limited information stored in the context trees. Obviously, for the longer contexts obtained from the linked lists, any PPM scheme can be implemented. Two examples have been mentioned before: the escape probabilities of PPMC [27] and PPM$*$ [12] require for each context a count of the number of different symbols following that context, and count exclusion, which even needs to know exactly which source symbols followed that context. Such techniques require in the context trees information referring to entire source symbols, rather than to the current or previous bits. This type of global information is not readily available in the decomposed context trees. On the other hand, while such techniques might greatly improve the overall performance of PPM, we are only interested in its performance on long repetitive strings. A good performance on long repetitive strings can be achieved by PPMA too, and therefore, we decided not to implement these techniques, but change the escape probability of PPMA instead.

For the experiments, we implemented several escape probabilities. The one that has been chosen in the end is a straightforward generalization of the PPMA-escape probability:

$$P_{\text{esc}}^{s,d} = \frac{\alpha_{PPM}}{C_s + \alpha_{PPM}}, \tag{7.15}$$

by introducing an additional parameter, α_{PPM}. Other escape probabilities, both simpler ones, and ones that should result in a more Lempel-Ziv '77-like behaviour have been investigated (see [57]), but they resulted in a poorer performance.

7.6 FIRST EVALUATION

7.6.1 THE REFERENCE SET-UP

Chapter 8 discusses the integration of all subsystems into one final design, and has a detailed evaluation of the performance of our final design. Here we will briefly evaluate the design to check whether it fulfils the requirements.

Both the parameters of the PPMA implementation and the parameters of the snake algorithm have to be determined. The PPM implementation has three parameters: D_{PPM}, C^* and α_{PPM}. If the maximum depth of the PPM algorithm, D_{PPM}, is higher than D then the PPM algorithm has to use the extended contexts. In that case, the second parameter C^* determines how many elements of the linked list in the pointer buffer are investigated at most. Finally, the parameter α_{PPM} can be used to modify the escape probability. The snake algorithm has two parameters that control its memoryless estimator: α_{sn} and B, the maximum count.

It is very difficult to find the set-up that achieves the exact global maximum, but our reference set-up achieves at least a local maximum. We decided to leave the set-up of the CTW algorithm unchanged, although the design will perform best if CTW and PPMA have a complementary behaviour, and this could partially be achieved be readjusting the CTW parameters. Our reference set-up uses the Huf-IV forward decomposition and the CTW set-up as described in Chapter 4 with a limited file buffer of 512 kBytes. The snake algorithm uses $\alpha_{sn} = 3$ and $B = 12$. The PPM algorithm uses $D_{PPM} = 25$, so it requires the additional pointer buffer for the extended contexts, and it investigates at most the first $C^* = 30$ entries of the linked list in this buffer. It uses $\alpha_{PPM} = 6$ for computing the escape probabilities. With this set-up the Calgary corpus is compressed to 737125 bytes.

7.6.2 THE SET-UP OF THE SNAKE ALGORITHM

Table 7.2: Changing the parameters of the snake algorithm.

			B		
			8 bytes	12 bytes	16 bytes
α_{sn}	2	Calg.	736909	737152	737396
		Cant.	2673912	2673187	2672864
	3	Calg.	736909	737125	737334
		Cant.	2671742	2671007	2670681
	4	Calg.	737140	737258	737424
		Cant.	2671139	2670219	2669890

The snake algorithm is characterized by the two parameters α_{sn} and B. We examined the performance of the total design for several combinations of these parameters on *two* corpora (the Calgary corpus and the Canterbury corpus, presented in Chapter 8) to prevent an over-adjustment of the parameters to our reference corpus. Table 7.2 shows for each corpus the compressed size in bytes. The results of the two corpora seem to contradict each other, therefore we decided to use $\alpha_{sn} = 3$ and $B = 12$ as a compromise. The value for α_{sn} is as anticipated, because there will be relatively few switches (roughly one every few hundred symbols), thus the parameter describing the transition sequence will be close to zero. For such sequences a value for α_{sn} greater than two is preferable. But, surprisingly, this set-up uses a very small value of B. This means that a

very adaptive estimator is chosen rather than an accurate one. It is difficult to match this highly adaptive mechanism of the snake algorithm with its original idea: the snake algorithm should try to track the best transition sequence. But it supports the ideas proposed in Section 6.6: clearly, the gap between the dominant state and the subordinate state cannot become very large at all. Apparently, the snake algorithm performs best if it can quickly adapt to local circumstances, and if, under the right circumstances, the subordinate state can become dominant within a few symbols.

7.6.3 THE SET-UP OF THE COMPANION ALGORITHM

Two settings of the PPMA algorithm deserve a closer investigation. First, the maximum depth D_{PPM} and the number of investigated contexts in the linked list C^* can be used to balance the trade-off between complexity and performance. In Table 7.3 the compression time of the Calgary corpus and the compressed size are collected for several combinations of these two parameters. The CTW algorithm alone compresses the entire corpus in 257 seconds to 745680 bytes. We choose $D_{PPM} = 25$ and $C^* = 30$ which seems a good trade-off. As expected, higher values of D_{PPM} and C^* result in an improved performance at the cost of a longer compression time. It is interesting to observe that increasing the depth D_{PPM} appears to be more efficient than increasing C^*. For example, $D_{PPM} = 30$ and $C^* = 50$ has a better performance than $D_{PPM} = 20$ and $C^* = 100$, while the compression time is about the same. Roughly said, a higher value of C^* results in more accurate and possibly more diverse estimates for long contexts, while a higher value of D_{PPM} could cause a higher selectivity and an even more dominant effect of the long repetitive strings on the estimate. This might indicate that the main contribution of the PPMA algorithm is indeed its superior performance on such strings.

Table 7.3: A performance-complexity trade-off.

	C^*					
	30		50		100	
	sec.	bytes	sec.	bytes	sec.	bytes
20	457	737498	471	737458	506	737344
D_{PPM} 25	462	737125	486	737069	531	736908
30	478	736980	502	736905	555	**736698**

The setting for α_{PPM} also needs some closer investigation. It has been chosen based on the results on the Calgary corpus (see Table 7.4), and although $\alpha_{PPM} = 7$ would be marginally better overall, $\alpha_{PPM} = 6$ has been chosen because it has a better performance on some files important for the strong performance requirement. Such a high value for α_{PPM} seems counter-intuitive, since it pulls the escape probability closer to one and effectively reduces the influence of the long matching contexts. But, in practice the situation is more complicated. Suppose so far the longest matching context is followed by the same symbol. This symbol will not only get a positive estimate for the longest matching context length, but also at every shorter context

length, because our implementation does not use count exclusion. Therefore, if the first other source symbol follows a much shorter matching context, then its code space will be reduced significantly by the series of escapes from the longest matching context to this context length. This is amplified by a small value for α_{PPM}. In that case the escape probability will be small, and the estimate of the PPMA algorithm will be dominated by the symbols following the longest matching context. At first sight, this appears to be exactly the desired behaviour, but it is not. A very common situation is that two long repetitive strings overlap: the first part of the second string is identical to the tail of the first. A PPMA algorithm with a very small α_{PPM} will estimate the symbols from the first string with a high probability. But the first source symbol that is not part of the first string will get a very small probability. This will make the PPMA algorithm most likely the subordinate state in the snake algorithm, and the distance between the states will be close to maximum. With the consecutive symbols, all part of the second repetitive string, PPMA will close the gap, and probably become the dominant state again, but it will take several symbols before the entire system can make use of the excellent PPMA performance on the second repetitive string. Choosing a greater value for α_{PPM} will reduce the gain the PPMA algorithm can achieve on long repetitive strings, but it will also reduce the penalty it has to pay when a repetitive string ends. The optimal value for α_{PPM} depends on many properties: the average length of the repetitive strings in the source sequence, and the average number of context depths it has to escape, among many others. Apparently a moderate value for α_{PPM}, that reduces both the gain achieved on a long repetitive string, as well as the penalty it has to pay after the string ends performs best.

Table 7.4: Changing the parameters of the escape probability.

α_{PPM}	4	5	6	7	8
Calgary corpus (bytes)	737529	737284	737125	**737029**	737034

7.6.4 EVALUATION

Details of the performance and the complexity of the reference set-up of the final design can be found in Table 7.5. The performance is compared to the performance of PPMZ-2. We only compare the performance on 12 out of the 14 files of the Calgary corpus, because PPMZ-2 preprocesses geo and pic. On seven files, especially the real text files and the object files, our final design performs considerably better than PPMZ-2: between 0.027 bits/symbol and 0.055 bits/symbol. On the five other files the performance is worse by only 0.027 bits/symbol or less. The entire corpus is compressed to 737125 bytes. The twelve files that can be compared are compressed to 633383 bytes by our final design and to 643140 bytes by PPMZ-2. This implementation meets the strong performance requirement.

The entire system, with the extended PPM algorithm, uses a hash table of 32 MBytes, a file buffer of 512 kBytes, a pointer buffer and several small tables. The pointer buffer has 524288 entries. In our implementation a pointer is 4 bytes, thus this buffer uses 2 MBytes. If we assume

Table 7.5: The performance of the final design.

	Snake, CTW and PPMA				PPMZ-2 v0.7	
	bytes	bits/sym.	sec.	kB/s	bits/sym.	diff
bib	24039	1.728	19.29	5.6	1.717	-0.011
book1	205614	2.140	106.00	7.1	2.195	+0.055
book2	137732	1.804	91.19	6.5	1.846	+0.042
geo	56638	4.425	12.68	7.9	4.097*	—
news	104072	2.208	57.55	6.4	2.205	-0.003
obj1	9768	3.634	5.14	4.1	3.661	+0.027
obj2	67698	2.194	43.48	5.5	2.241	+0.047
paper1	14522	2.185	8.97	5.8	2.214	+0.029
paper2	22170	2.158	12.30	6.5	2.185	+0.027
pic	47104	0.734	56.50	8.9	0.480*	—
progc	11048	2.231	7.48	5.2	2.258	+0.027
progl	13182	1.472	12.58	5.6	1.445	-0.027
progp	9069	1.469	9.63	5.0	1.450	-0.019
trans	14469	1.235	18.82	4.9	1.214	-0.021

that all small tables, all registers, and the program code itself fit in 512 kBytes, then the entire system uses a hash table of 32 MBytes plus an additional 3 MBytes. Thus the total memory usage is less than 10 % above the 32 MBytes from the requirements. This seems an acceptable overhead.

Table 7.3 shows that the speed of the entire system depends greatly on D_{PPM} and C^*. The system in its reference set-up compresses the corpus in about 462 seconds, while CTW alone needs only 257 seconds. The entire corpus is 3068 kBytes, thus the average compression speed is only 6.6 kBytes/sec. This is only two thirds of the required 10 kBytes/sec. The highest compression speed, measured on pic, is only 8.9 kBytes/sec. (see column kB/s in Table 7.5). It is clear that the speed requirement is not met.

Thus the entire system, CTW, PPMA and the snake algorithm, achieves the strong performance requirement. It uses less than 35 MBytes, and this seems a reasonable overhead, so it meets the memory requirement. But the compression speed is only 6.6 kBytes/sec. It does not meet the compression speed requirement.

8

INTEGRATION AND EVALUATION

This chapter discusses the integration and the final evaluation of the design. So far, this thesis focused mostly on the workings of the individual subsystems, with only a minimum description of the interaction between the subsystems. This chapter provides an overview of the integration of the entire design from four different perspectives: the algorithmic perspective, the arithmetic perspective, the data structure perspective and the implementation perspective. Next, the modularity of the subsystems is considered. The modularity has been achieved to a great extent: during the implementation of the final design we could easily exchange different implementations of most subsystems, and most subsystems can be re-used in other designs. The only exception is the companion algorithm which relies too much on the CTW implementation. Finally, the final design is evaluated. We present two set-ups, a set-up consisting of the forward decomposition, CTW and the arithmetic coder. This set-up satisfies the weak performance criterion, the speed and the complexity criteria. In the second set-up the snake algorithm and the companion algorithm are added. This design no longer meets the compression speed criterion (it is about 34 % too slow), but on the other hand it might deliver the best compression performance to date.

8.1 SYSTEM INTEGRATION

So far in this thesis we have investigated the design in almost every aspect. Overview Chapter 2 describes the entire design process, while technical Chapters 3 to 7 give a very detailed description of every subsystem. Although all technical information is available in these chapters, some system-wide implementation issues are spread over several chapters. Here, we would like to highlight some system integration considerations. We chose the four different perspectives, or views, that seem most relevant for the final design. The first view is the algorithmic perspective shown in Figure 8.1(a) and discussed in Section 8.1.1. Two aspects are important in this view. First, it clearly shows how the three step design process resulted in the final design, consisting

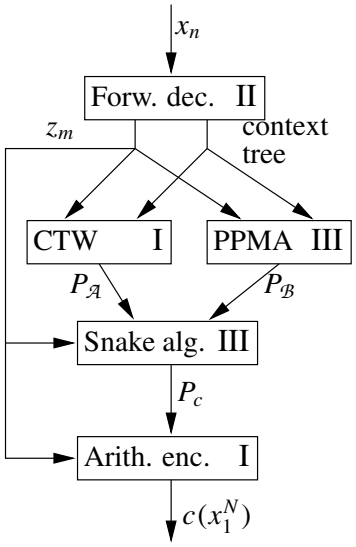
of five subsystems. The second aspect of this view involves a major design decision: some subsystems operate on bits, others process entire source symbols. How is this handled? Section 8.1.2 and Figure 8.1(b) discuss the arithmetic perspective. For a lossless compression program, the arithmetic and the alignment of the probabilities communicated between the different subsystems is simply crucial, otherwise a cross-platform compatibility can never be guaranteed. Section 8.1.3 and Figure 8.1(c) treat the design from a data structure perspective. Together with the arithmetic, the data structures are the main design elements. The fourth view is the implementation or deployment perspective, discussed in Section 8.1.4 and shown in Figure 8.1(d). This section simply describes the integration of all subsystems into one program and the communication between the subsystems. In short, it discusses the main loop. Closely related to the integration of the subsystems is the modularity of the design. This is examined in Section 8.1.5.

8.1.1 THE ALGORITHMIC PERSPECTIVE

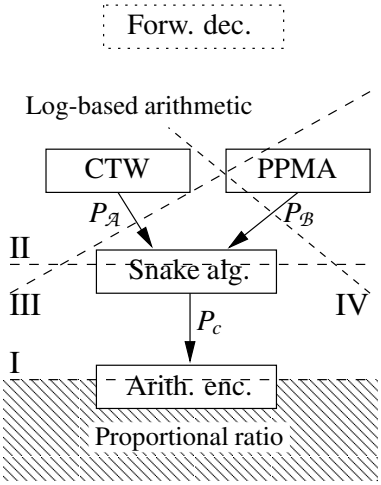
Figure 8.1(a) shows the algorithmic perspective of our final design. The algorithmic aspects of the design and of the subsystems have been described in detail in Chapter 2 and in Chapters 3 to 7, respectively. Ultimately, the design process consisted of three steps, resulting in a final design with five subsystems. The Roman numbers in the boxes in Figure 8.1(a) indicate the step of the design process in which that subsystem was added to the design. The original intention was that the design would be finished after a single design step, resulting in a design consisting of an arithmetic coder (see Chapter 3), and the CTW algorithm (Chapter 4). After evaluation, it was decided to add the forward decomposition (Chapter 5) in a second step for complexity reduction. Finally in a third step, the snake algorithm (Chapter 6) and the companion algorithm (Chapter 7) completed the final design for performance improvement. As discussed before, a design process in several steps has its advantages, e.g. easier integration, but also its disadvantages. It has complicated the algorithmic side of the design somewhat, especially the design of the companion algorithm (see Chapter 7).

One important algorithmic decision is the choice of the source alphabet for the various subsystems. For reasons of complexity, ease of implementation, and especially performance, we decided to decompose the source symbols into bits and use a "binary" CTW algorithm[1]. The CTW algorithm has been designed from the start to operate on bits, and the encoder can estimate the probability of every bit of a source symbol (given its context), independently of the other bits. This decomposition also simplifies the design of the arithmetic coder. The design decision is further supported by the special forward decomposition, which reduces the memory complexity, increases the speed and improves the performance at the same time. The other parts of our final design, the snake algorithm and the companion algorithm are designed to operate on entire source symbols. The snake algorithm interfaces with the CTW algorithm and the companion algorithm on one hand and with the arithmetic coder on the other hand. It has to operate on the same alphabet size as both the CTW and the companion algorithm. Since the CTW algorithm cannot be easily converted to processing entire source symbols, both the companion and the

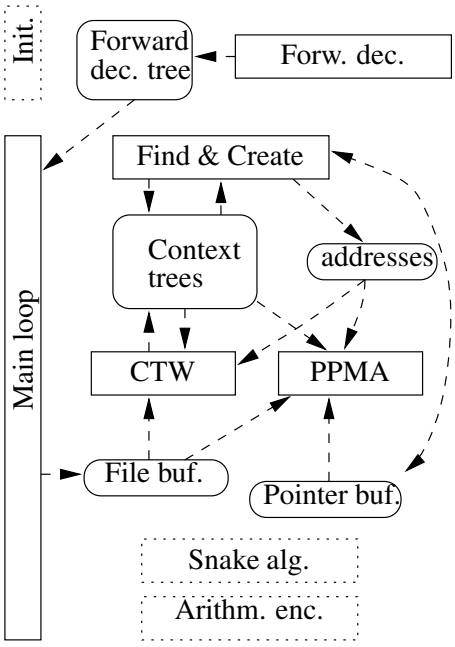
[1] Although our CTW algorithm decomposes the source symbols in order to be able to use binary estimators, the context symbols are not decomposed and belong to the original source alphabet.



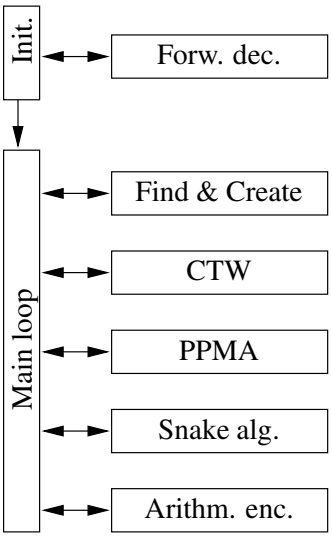
(a) The algorithmic perspective



(b) The arithmetic perspective



(c) The data structure perspective



(d) The implementation perspective

Figure 8.1: The design viewed from different perspectives.

snake algorithm have to be decomposed. But because these algorithms originally operate on entire source symbols, they cannot estimate the probability distribution of each bit independently, and the decomposition process will be more difficult. This has been discussed in detail in Chapters 6 and 7, and will briefly be highlighted in the section on the implementation perspective, Section 8.1.4.

8.1.2 THE ARITHMETIC PERSPECTIVE

The arithmetic perspective of the design is shown in Figure 8.1(b). If the design uses all subsystems, then CTW and PPMA both supply a probability distribution to the snake algorithm and the snake algorithm supplies a probability distribution to the arithmetic coder. Not shown is the alternative situation in which CTW supplies its probability distribution directly to the arithmetic coder. Many computations in these subsystems use real-valued numbers. Clearly, floating point numbers cannot be used to represent these values and consequently the subsystems have to use their own representation for these numbers. While integrating these subsystems, we have to ensure that the snake algorithm and the arithmetic coder can handle the representation in which they receive their input probability distributions.

The implementation of the CTW algorithm was designed in the first step of the design process. It uses only additions, multiplications and divisions in its computations. Therefore, we decided to use a fixed-point log-representation for these numbers in the CTW algorithm (see Chapter 4). In this step of the design process CTW communicated directly to the arithmetic coder. The arithmetic coder requires for every bit the ratio between the probability of a zero for that bit, and the probability of a one. It represents these probabilities by two integers that are proportional to the probabilities. Thus at some point the probabilities have to be translated from their fixed-point logarithmic representation used by CTW to the proportional ratio used inside the arithmetic coder. This translation takes place at the point indicated by line I in Figure 8.1(b). In the third step of the design process, PPMA and the snake algorithm have been added to the system. Now also for these two new modules a good representation for the real-valued numbers has to be found. But for this decision not only the most efficient implementation of the computational work in these subsystems counts, but also the number of translations necessary in the final design, because each translation costs time and accuracy. If possible, it would be best if these two new subsystems would either use the same fixed-point log-representation as CTW (resulting in only one translation, at line I), or the proportional representation used by the arithmetic coder (only one translation at line III). The applicability of the proportional representation is rather limited, and this representation seems only useful inside the arithmetic coder. Therefore, the translation indicated by line I appears to be unavoidable. If either PPMA or the snake algorithm would not be able to compute with the fixed-point log-representation of CTW, then extra translations between CTW and the snake algorithm (line III), between PPMA and the snake algorithm (line IV), or both (line II) would be necessary. Fortunately, PPMA can be designed such that it uses the same fixed-point log-representation as CTW (see Chapter 7), and as a result the snake algorithm can be easily implemented with this representation too. Hence, almost the entire final design uses the fixed-point log-representation, and the only necessary translation takes place inside the arithmetic coder (line I).

8.1.3 THE DATA STRUCTURE PERSPECTIVE

The design from a data structure perspective is shown in Figure 8.1(c). The different subsystems communicate with each other through the main loop and through the common data structures. Figure 8.1(c) shows these interactions, with the data structures in the boxes with rounded corners and the algorithms in the boxes with sharp corners.

The most important data structure is the one that contains the context trees. This structure has been created for the CTW algorithm during the first step of the design process. One of our objectives was to keep the data structure and all necessary search and update algorithms completely separated from the CTW computations. This has the advantage that different data structures and their corresponding search and update algorithms could be plugged into the system without changing the computational part of CTW. In such a system CTW is split in a structural part and a computational part. Only the structural part can access the real data structure. In such a system, the structural part first finds the records in the data structure, and places the information from these records in a common data structure. The computational part reads the information from this structure, and after coding the next bit, it updates the information in the common structure. Next, the structural part would read the updated information from the common structure and make the appropriate changes in the real data structure.

In the real implementation of the final design the computational part (denoted by 'CTW' in Figure 8.1(c)) and the structural part of CTW (denoted by 'Find & Create') have not been separated completely. In practice, first the structural part searches for, and possibly creates, the records corresponding to the nodes and the leaf on the context path. But then, instead of passing on *the information* in the records, it simply collects *the memory addresses* of these records in a table. Next, the computational part of CTW uses these addresses to obtain its information directly from the records in the actual data structure. Therefore, it requires exact knowledge of the structure of the records. Furthermore, after coding the new bit, the computational part of CTW also changes the records in the data structure directly. This implementation was chosen, simply because it is faster. In the third step of the design process, the companion algorithm is added to the design. It also needs to have information from the context trees. But since it needs to use *the same* records as CTW, it simply uses the table with memory addresses already generated by the structural part of CTW. As a result also the companion algorithm has to have knowledge about the exact structure of these records. In order to prevent ambiguities in this data structure shared by CTW and the companion algorithm, only CTW is allowed to make changes to the records.

The structural part of CTW and the data structure for the context trees have been designed in the first step of the design process based on two main criteria. First, the records should be as small as possible, and secondly, the structure should allow an efficient search for the consecutive nodes on a context path in a context tree. The data structure has been implemented as 6 bytes records in a 32 MBytes hash table. Hashing is an ideal choice for our case, because it is fast and requires only very little information (less than two bytes) in each record to maintain the structure. For other implementations it might be less useful because it also imposes some limitations. For example, one cannot easily find all children of a node, which is necessary for algorithms that apply some form of tree pruning. The CTW information in each node can be packed in four

bytes, resulting in records of six bytes each. As mentioned before, as a result of the three step design process, the companion algorithm is not allowed to enlarge the records again, so it cannot store its own information in the records, and consequently, its performance is crippled.

There are many other, smaller, data structures in the final design. During the initialization the forward decomposition constructs the forward decomposition tree which is passed on to the main loop through a data structure. During coding CTW uses a 512 kBytes file buffer. This buffer is maintained by the main loop. If the companion algorithm is activated, it will use this buffer too. The companion algorithm will also use a pointer buffer with 512k entries, one for each symbol in the file buffer. This buffer will be maintained by the structural part of CTW ('Find & Create'). Finally, most of the subsystems have their internal data structures. E.g., the snake algorithm uses a small data structure consisting of two states, each with a counter and a probability. These data structures have been omitted from Figure 8.1(c).

8.1.4 THE IMPLEMENTATION PERSPECTIVE

The implementation of the subsystems has been described in Chapters 3 to 7. But these subsystems have to be integrated into one final design. Here we will examine the entire design from an implementation perspective (see Figure 8.1(d)). From this figure it is clear that there is no direct communication between the subsystems: they exchange information only through the main loop and via common data structures. This is necessary to obtain a high degree of modularity in the final design.

The main loop is the heart of the final design. It activates each module at the correct moment during coding, and it administrates all necessary information, e.g. the current context, the file buffer, and some general performance statistics, and it passes this information on to the subsystems. Furthermore, it keeps track of which node in the forward decomposition tree is active, and it provides the searching routines with the correct starting address of the corresponding context tree.

The data compression program starts with an initialization step. In this step it activates the subsystem that will construct the forward decomposition, it will reset all relevant parameters and compute several tables. After this, the main loop will be started, and it has to activate the different subsystems in the correct order. There are two types of subsystems: those that work inherently on bits, and those that work on entire source symbols. The main loop has to work at least at the same processing pace as that subsystem that works on the smallest entities. Therefore, the main loop processes the individual bits.

Both the arithmetic coder and CTW process bits. The arithmetic coder requires at only one point interaction with the main loop. As soon as the probability for the next bit has been estimated, either by CTW directly or by the snake algorithm, it will be activated to code the next bit. CTW is split in a structural part ('Find & Create') and a computational part ('CTW'). The structural part is started first to find and possibly create the records on the context path. Next the computational part is started to compute the estimated probability, and to compute the updated records for both the case the next bit is a zero and a one. After the bit is coded, the structural part is activated a second time to update the data structure containing the context trees.

The snake algorithm and PPMA originally process entire source symbols instead of single

bits. These two algorithms have been integrated into the final design by decomposing them. But such decomposed algorithms differ in a fundamental way from algorithms, like CTW and the arithmetic coder, that have been designed to process bits. The two algorithms that have been designed from the start to process bits can encode each bit of a source symbol independently from the estimates for the other bits, while for the algorithms that have been designed for entire source symbols, and that have been decomposed afterwards, the encoding of later bits in a source symbol depends on the estimates computed for the previous bits of that symbol. The algorithms that have been decomposed later interact with the main loop as follows. Before the first bit of a new source symbol is coded, the weights for the internal states (the snake algorithm has two states, the PPMA algorithm at most D_{PPM}) of these algorithms have to be computed. These weights should be used to weight the estimated probabilities of the entire source symbol. Next, before a bit is coded, the algorithms are activated to give an estimate of the probability distribution of this bit. Because these algorithms cannot treat the bits independently, after this bit has been coded, the algorithms have to be activated again. The PPMA algorithm explicitly stores the relevant information for the next bit in a structure of correction factors, while the snake algorithm will update its weights to account for the dependency between the bits of a source symbol.

Now the main loop with the snake algorithm and the extended PPM algorithm can be described in a high level pseudo-code with the following steps.

- The encoder will read the file, construct the forward decomposition and store sufficient information in the compressed file for the decoder. The decoder will read this information from the compressed file and construct the forward decomposition.
- Write or read the first D uncoded symbols.
- For all x_n, with $n = D+1, D+2, \ldots, N$, do:
  - ⋆ Select the root node of the forward decomposition.
  - ⋆ Suppose $x_n = z_1, \ldots, z_M$. For all z_m, with $m = 1, \ldots, M$ do:
    - ▷ Find and create the path in the context tree corresponding to the current node in the forward decomposition.
    - ▷ If $m = 1$, this is the first bit, then compute the weights of the snake algorithm and the weights of the PPM algorithm.
    - ▷ Compute the estimated probability for z_m with PPM, $P_{\mathcal{B}}(z_m|x_1^{n-1}, z_1^{m-1})$.
    - ▷ Compute the estimated probability for z_m with CTW, $P_{\mathcal{A}}(z_m|x_1^{n-1}, z_1^{m-1})$.
    - ▷ Compute the estimated probability for z_m with the snake algorithm, $P_c(z_m|x_1^{n-1}, z_1^{m-1})$.
    - ▷ Encode or decode bit z_m with $P_c(z_m|x_1^{n-1}, z_1^{m-1})$.
    - ▷ Update records in the context tree given the value of z_m.
    - ▷ Update weights in the snake algorithm given the value of z_m.
    - ▷ Update the correction factors $\delta(.)$ of the PPM algorithm given the value of z_m.
    - ▷ Select the correct child node in the forward decomposition.

- ⋆ Update file buffer. The pointer buffer will be updated by the subroutine that finds and creates the context path in the context trees.

8.1.5 MODULARITY

Modularity works in two directions. From one direction, modularity deals with the ability to replace one subsystem of the final design by a similar subsystem. From the other direction, modularity is concerned with the ability to use the subsystems designed for this project in other designs. During the design process we focused especially on the first direction. We intended to realize this by restricting the subsystems in two ways. First each subsystem is only allowed to communicate with the other subsystems through the main loop. As a result, only the main loop has to know exactly how and when to activate each subsystem. Secondly, in case the subsystems have to share information this should be handled through predefined common data structures. In this way the internal data structures of each subsystem are hidden from the other subsystems. As we have seen in the previous sections, these two restrictions have been realized only partially. Let us briefly investigate the degree of modularity achieved by each subsystem.

The arithmetic coder: The arithmetic coder is the most independent part of the entire system: it is hooked up in the main loop at only one point, and it doesn't share any data structures with the other subsystems. The only implementation specific property of our arithmetic coder is the necessary translation of the input probability distributions in fixed-point log-representation. While developing the final design we could plug-in either the Rubin coder or the WNC coder (see Chapter 3), without any change to the main loop. By removing the implementation specific translation of the numbers inside the arithmetic coder, both coders can be easily used in other designs too.

The forward decomposition: The forward decomposition is activated only once, during initialization, and it shares one predefined common data structure with the main loop. This makes it very easy to design different forward decompositions and plug them in at will. While developing the final design we could choose between any of the four Huffman decompositions (see Chapter 5) and the straightforward fixed decomposition (the ASCII decomposition), without any change to the main loop. On the other hand, although this subsystem can be easily incorporated in other designs, it is a very application specific subsystem that may have little use outside our design.

The CTW algorithm: The implementation of the CTW (see Chapter 4) algorithm is split into two parts: the structural part (dealing with the data structures and search and update routines) and the computational part (dealing with the actual probability computations). These two parts have not been separated as strictly as intended, but still different structural parts or different computational parts could be plugged-in the design with only marginal changes. This particular feature has not been exploited in our design because CTW was a requirement of the final design, therefore the computational part is fixed and the current

implementation of the structural part appears to be the most efficient solution at the moment. Together these two subsystems achieve a higher degree of modularity. They can be easily transferred in full to other designs, although they might need an additional translation step that converts the resulting estimated probability distributions in the fixed-point log-representation to a different representation of these floating point numbers.

The companion algorithm: The companion algorithm, PPMA (see Chapter 7), is symbol-oriented, thus it interacts with the main loop at three places. Because it could not use a data structure of its own due to the memory restrictions, it shares several data structures with the CTW subsystem, especially the context trees and the small table with the addresses of the records on the context path. This clearly restricts the modularity of this subsystem. On one hand, during the implementation of the final design we could plug-in different variations of the companion algorithm. On the other hand, this implementation of the companion algorithm relies completely on the structural part of CTW (and partially on the computational part for updating the counts). So it cannot be ported to other designs without porting the structural part of CTW too, and porting both just to get a PPM implementation is inadvisable, because designing a new data structure dedicated to PPM will result in a more efficient design.

The combining algorithm: Because the snake algorithm (see Chapter 6) is symbol-oriented, it is tied in the main loop at three places. It expects its input probability distributions in the fixed-point log-representation and it also outputs a probability distribution in the fixed-point log-representation. It does not share any common data structures with the other subsystems. As a result, any combining algorithm that operates in the fixed-point log-domain can be used. During the implementation of the final design we could choose between the snake algorithm, the switching algorithm and the reduced complexity algorithm (which has not been discussed here), without any changes to the main loop. The fixed-point log-representation is the only complication for using this subsystem in other designs. But, if necessary, this can be solved by some additional translation steps.

In summary, the modularity has been achieved to a high degree. For most subsystems we had alternatives that could, almost effortlessly, be plugged into the final design. Reversely, many of these modules can be used in other designs as well. The only real exception is the companion algorithm. It depends completely on the structural part of CTW.

8.2 EVALUATION

8.2.1 OUR FINAL DESIGN

The final design will now be evaluated. The project description contains four performance measures: performance, compression speed, complexity and universality. In theory the final design can handle any alphabet size, because the forward decomposition will convert the symbols into bits. In practice, our implementation can use alphabets of up to 256 different symbols. This is

sufficient, since most computer systems store their information in bytes. Thus the universality requirement is met by our design. The other three requirements are part of a trade-off. An improvement in performance might result in an increase in complexity or a decrease in speed. Also, complexity and speed can be exchanged to some degree. In this thesis we propose two trade-offs.

1. The first system, called "**CTW**" here, consists of a CTW algorithm (with context depth 10, see Chapter 4), a forward decomposition (the Huf-IV decomposition, see Chapter 5), and an arithmetic coder (the WNC coder, see Chapter 3). This system uses one hash table of 32 MBytes, a file buffer of 512 kBytes, and some smaller tables. In total the memory complexity is less than 33 MBytes.

2. The second system, called "**Snake**" here, has two additional components: a snake algorithm that switches between two universal source coding algorithms (see Chapter 6), and a companion algorithm (PPMA with context depth 25). This system requires an additional pointer buffer of 2 MBytes, and a few additional tables. The total memory complexity of this system is about 35 MBytes. This system will also be considerably slower than the CTW system, because it runs two data compression algorithms in parallel.

8.2.2 THE COMPETING ALGORITHMS

The performance of our design should be competitive with other state-of-the-art data compression algorithms. In the project definition (see Chapter 2) we decided to compare the performance to both "state selection" [10] (see Table 4.11) and PPMZ-2 v0.7 [9]. Here we will compare our design only to PPMZ-2 because it is a more recent algorithm and as a result it has in practice a performance that is superior to "state selection", and all other non-CTW based data compression algorithms published in the proceedings of the DCC up to and including 2001. It is also important to mention that, at this moment, there is an interesting program called "rk" [42] that is considerably faster than PPMZ-2 (and Snake), uses less memory and achieves a very good performance on several files. Unfortunately, there is little documentation on the internal workings of this program. Experiments on a wide range of files show that the performance of this algorithm varies considerably, which might suggest that it is over-adjusted to specific types of files, e.g. English texts. Therefore, to get a better indication of the performance of this program, we will only assess it on the longest files of the Canterbury corpus.

8.2.3 THE PERFORMANCE ON THE TWO CORPORA

The performance and the compression speed of our design will be measured on two corpora: the Calgary corpus [7] (see Table 4.5 in Chapter 4) and the Canterbury corpus [5] (see Table 8.1). The Canterbury corpus consists of two parts. The first 11 files in Table 8.1 form the small corpus, while the large corpus has three additional long files.

The performance of CTW and Snake on the two corpora, compared to PPMZ-2, can be found in Table 8.2. For both corpora the column average (favouring small files) as well as the average weighted on file length (favouring large files) are computed. Note that the results of PPMZ-2 on

the Calgary corpus are taken from the home-page [9] and we may assume that PPMZ-2 is used in its optimal set-up, while the results on the Canterbury corpus have been obtained by running the program ourselves in the default set-up. Running PPMZ-2 in default mode on the Calgary corpus would only change the results for geo and pic. This explains, why our tables give different results for PPMZ-2 on the one file that occurs in both corpora (as pic and ptt5). Thus PPMZ-2 can gain considerably in performance on some files if it is run in an optimal set-up. In this respect, we have to realize that it is possible that such an optimal set-up of PPMZ-2 performs better e.g. on the file kennedy.xls than reported in Table 8.2. Unfortunately, we were not able to run PPMZ-2 on the three additional files of the larger Canterbury corpus.

Table 8.1: The files in the Canterbury corpus.

name	size	type	description
alice29.txt	152089	text	book by Lewis Carroll, "Alice's Adventures in Wonderland"
asyoulik.txt	125179	text	play by Shakespeare
cp.html	24603	text	a HTML file with data compression related links
fields.c	11150	text	C source code
grammar.lsp	3721	text	Lisp source code
kennedy.xls	1029744	binary	Excel spreadsheet
lcet10.txt	426754	text	technical document
plrabn12.txt	481861	text	English poetry by John Milton, "Paradise Lost"
ptt5	513216	binary	bitmap of a black and white picture (same as pic)
sum	38240	binary	SPARC executable
xargs.1	4227	text	GNU manual page
E.coli	4638690	binary	complete genome of the E. Coli bacterium
bible.txt	4047392	text	the King James Bible
world192.txt	2473400	text	the CIA World Factbook 1992

The numbers in italics in Table 8.2 mark the results that do not meet the strong performance requirement. Clearly the CTW has a superior performance on the text files (book1, book2, paper1, paper2, alice29.txt, asyoulik.txt, lcet10.txt and plrabn12.txt) compared to PPMZ-2. On the Calgary corpus, CTW loses on almost all other files. On some files it doesn't meet the strong performance criterion. But, if we consider only the 12 files that can be compared in fairness with PPMZ-2 (we excluded pic and geo because PPMZ-2 preprocesses these files), then book1 and book2 are by far the largest files, and the better performance of CTW on these files dominates the performance on all 12 files. As a result, CTW compresses these files to only 641945 bytes, while PPMZ-2 requires 643140 bytes. The results on the Canterbury corpus are even more difficult to compare. The file kennedy.xls is by far the longest file of the corpus. The huge difference in performance on this file between PPMZ-2 on one hand and CTW and Snake on the other hand will mask the performance on the remaining files completely. If we also exclude file ptt5, then CTW compresses the nine other files to 322592 bytes, and PPMZ-2 compresses them to 327313

Table 8.2: The performance of the final design.

	PPMZ-2	CTW		Snake	
Calgary	bits/sym.	bits/sym.	diff.	bits/sym.	diff.
bib	**1.717**	1.757	+0.040	1.728	+0.011
book1	2.195	2.145	-0.050	**2.140**	-0.055
book2	1.846	1.823	-0.023	**1.804**	-0.042
geo	**4.097**	4.417	—	4.425	—
news	**2.205**	2.253	*+0.048*	2.208	+0.003
obj1	3.661	3.640	-0.021	**3.634**	-0.027
obj2	2.241	2.253	+0.012	**2.194**	-0.047
paper1	2.214	2.212	-0.002	**2.185**	-0.029
paper2	2.185	2.176	-0.009	**2.158**	-0.027
pic	**0.480**	0.736	—	0.734	—
progc	2.258	2.264	+0.006	**2.231**	-0.027
progl	**1.445**	1.517	*+0.072*	1.472	+0.027
progp	**1.450**	1.549	*+0.099*	1.469	+0.019
trans	**1.214**	1.299	*+0.085*	1.235	+0.021
Column av.	**2.086**	2.146	+0.060	2.116	+0.030
Weighted av.	**1.910**	1.961	+0.051	1.939	+0.029
Canterbury					
alice29.txt	2.059	2.028	-0.031	**2.012**	-0.047
asyoulik.txt	2.308	2.275	-0.033	**2.264**	-0.044
cp.html	**2.156**	2.218	*+0.062*	2.190	+0.034
fields.c	1.895	1.898	+0.003	**1.860**	-0.035
grammar.lsp	**2.270**	2.335	*+0.065*	2.331	*+0.061*
kennedy.xls	1.364	**0.833**	-0.531	0.852	-0.512
lcet10.txt	1.798	1.778	-0.020	**1.758**	-0.040
plrabn12.txt	2.195	2.154	-0.041	**2.149**	-0.046
ptt5	0.751	0.736	-0.015	**0.734**	-0.017
sum	2.536	2.458	-0.078	**2.392**	-0.144
xargs.1	**2.820**	2.905	*+0.085*	2.898	*+0.078*
Column av.	2.014	1.965	-0.049	**1.949**	-0.065
Weighted av.	1.568	1.358	-0.210	**1.358**	-0.210

Table 8.3: The compression speed of the final design.

	CTW		Snake	
	time sec.	speed kBytes/sec.	time sec.	speed kBytes/sec.
Calgary corpus	257	11.9	462	6.6
Canterbury corpus	189	14.5	335	8.2

bytes. Therefore, we may conclude that CTW meets the weak performance criterion.

If we compare the performance of Snake with CTW, then we see something interesting. On two files geo and kennedy.xls, CTW performs better than Snake. As explained in Chapter 6 this is possible, but in practice it should occur very rarely. Snake has a superior performance on all other files and as a result, it performs better than CTW on both corpora. It compresses the Calgary corpus to 737125 bytes compared to 745680 bytes with CTW, and the Canterbury corpus to 476983 bytes, compared to 477016 bytes with CTW. The difference between the latter two results is so small because Snake has a slightly poorer performance on kennedy.xls: it is a relatively small loss, but due to the size of this file, it is a big loss in absolute numbers. Snake has a much better overall performance: it easily meets the strong performance criterion on the Calgary corpus. On the Canterbury corpus we see another interesting issue with CTW and Snake. Both algorithm do not meet the strong performance criterion on grammar.lsp and xargs.1. Also the performance on cp.html is very weak, and only Snake barely meets the criterion. These three files are all very small. Therefore we can conclude that CTW and Snake may have a superior performance on longer files, thus they have a better convergence in the long term, but PPMZ-2 converges faster and as a result performs better on shorter files. Experiments with other competitive data compression algorithms confirm this notion: if the file is long enough then in general CTW and Snake will achieve a lower compression rate than other algorithms, but they learn slower, and as a result the performance on short files (up to several kBytes), is insufficient.

Table 8.3 summarizes the compression speed of the final design. Note that we ported CTW and Snake to a regular PC to compare its speed to PPMZ-2. On this PC both CTW and Snake are faster than PPMZ-2: PPMZ-2 requires about 50 % more compression time than Snake. CTW achieves an average compression speed of more than 10 kBytes/sec and meets the compression speed criterion, while Snake is too slow.

Discussion. Chapter 2 discussed the measure speed. As alternative measure the number of computations per bit was proposed. Profiling information from set-up CTW shows that about 40% of the compression time is spent on actually performing the CTW computations, about 40% of the time is spent on tree maintenance and 6% is used by the arithmetic coder (the remaining time is used by initialization, and lost on overhead). This shows, that counting only the number of computations will not accurately estimate the speed of the system.

This does not answer the question of scalability of the compression speed results to other (more common) platforms. Maybe our reference computer has unusually fast memory access, from which our

Table 8.4: The performance on the additional files of the large Canterbury corpus.

	rk1.02a		CTW		Snake	
	bytes	bits/sym.	bytes	bits/sym.	bytes	bits/sym.
E.coli	1128164	1.946	1119490	1.931	**1114941**	**1.923**
bible.txt	711116	1.406	708085	1.400	**690920**	**1.366**
world192.txt	**384088**	**1.242**	396734	1.283	388161	1.256

Table 8.5: The compression speed on the additional files of the large Canterbury corpus.

	CTW		Snake	
	time	speed	time	speed
	sec.	kBytes/sec.	sec.	kBytes/sec.
E.coli	525	8.6	659	6.9
bible.txt	383	10.3	727	5.4
world192.txt	226	10.7	527	4.6

implementation benefits unexpectingly. Recall that CTW uses a 32 MBytes hash table, through which it will jump almost randomly. This will most probably cause many cache misses, putting more stress on the memory bus. To verify that our design is not over-adjusted to this one platform, we ran the set-up CTW on a 233 MHz PentiumII PC (SPECint95 of 9.38) with 128 MBytes of memory (running FreeBSD). Based on SPECint95 alone, CTW should compress the Calgary corpus in 198 seconds and the Canterbury corpus in 144 seconds on this system. It actually compresses the Calgary corpus in 199 seconds, and the Canterbury corpus in 152 seconds. Thus, the increase in speed scales very well with the increase in SPECint95.

8.2.4 THE PERFORMANCE ON LARGE FILES

Finally, we investigate the performance on the three additional files of the large Canterbury corpus. The performance results for both CTW and Snake are collected in Table 8.4 and the measured compression speeds are collected in Table 8.5. Unfortunately, these results cannot be compared to PPMZ-2, since it required more than the 100 MBytes of free memory available on our PC. Therefore we decided to compare the performance to rk. rk has shown an excellent performance on English texts thus we expect a very good performance on the King James Bible and the CIA World Factbook. The DNA sequence consists of only 4 symbols, therefore most efficient data compression programs will be able to compress this file to about 2 bits/symbol, but the models used by the current data compression algorithms do not seem to match the models behind DNA very well. As a result, only few data compression algorithms are able to gain even slightly beyond this bound.

Both CTW and Snake use a 64 MBytes hash table, instead of a 32 MBytes hash table, for

world192.txt and bible.txt. On all three files Snake outperforms CTW. On the two test files, world192.txt and bible.txt, Snake gains 0.027 bits/symbol and 0.034 bits/symbol, respectively. This is similar to or even higher than the gains recorded on the smaller text files where the gain ranges between 0.005 bits/symbol (on book1 and plrabn12.txt) and 0.027 bits/symbol (on paper2). This shows that Snake improves the performance continuously throughout the entire text file, and not only at the start. This indicates that even on long texts, CTW is not able to handle long repetitive strings well. This could be caused by contexts that are sometimes a part of the same long repetition, and sometimes not. As expected rk performs really well on both text files. It is better than both CTW and Snake on world192.txt. But surprisingly, CTW is already better than rk on bible.txt, and Snake is considerably better. This could be caused by the rather atypical formatting of this text file and the slightly unusual English used in the King James Bible. This supports the idea that rk might be over-adjusted. On E.coli too, CTW and Snake have an excellent performance: they both perform better than rk, and are significantly below the 2 bits/symbol which is a very good result.

8.2.5 THE FINAL ASSESSMENT

In conclusion, the user can choose between two set-ups of our final design: CTW or Snake. Both programs are universal. CTW runs in about 33 MBytes, it compresses the corpora with more than 10 kBytes/second, and on average it compresses as good as the best state-of-the-art competitive data compression programs. But in general it achieves this by gaining significantly on medium length and long text files, while it loses slightly on different types of files. It meets the weak performance criterion. Snake uses less than 35 MBytes of memory. This is an overhead of not more than 10 %. It reaches an average compression speed of only 6.6 kBytes/second on the Calgary corpus. This is a third too slow. It compresses all files within 0.04 bits/symbol of the best competitive algorithms, except for really short files. Overall, it performs better on the vast majority of the files, and significantly better on text files. Except for short files, it meets the strong performance criterion, and, performance-wise, it is one of the best data compression algorithms at this moment (end 2002).

We believe that this combined final design successfully completes the project.

9

CONCLUSIONS AND RECOMMENDATIONS

9.1 CONCLUSIONS

THIS thesis describes the design and implementation of a data compression program based on the CTW algorithm. First, the most important specifications of the final design have been condensed into three simple, measurable, requirements: compression performance, compression speed and memory complexity. Our starting point was a streamlined implementation of the CTW algorithm and an arithmetic coder. Evaluation of this first design revealed two shortcomings: speed and performance. By introducing a forward decomposition, which is constructed during a "pre-scan" of the source sequence, the speed could be hugely increased (by up to 50 %). Next, the snake algorithm and the companion algorithm PPMA were introduced which improved the performance such that the new design met the strong performance requirement (at the cost of a reduction in speed, such that the speed requirement is not satisfied anymore). These steps, the algorithms, their implementation and their evaluation have been thoroughly discussed throughout this thesis.

Besides this thesis, the product of the project is a data compression program, written in ANSI C. This program has been successfully compiled and used on multiple platforms. Experiments show that the program works reliable, and that files compressed on one system can be decompressed on an other system. The program can handle any type of file, as long as the length of the file does not exceed 16 MBytes (after that several 24-bit counters will overflow). To overcome the dilemma of choosing between the speed requirement and the strong performance requirement, we decided to let the user choose between two set-ups of the final design: CTW and Snake.

Set-up **CTW** uses 32 MBytes of memory, it achieves a compression speed of about 10 kBytes per second on our reference computer system, and on average, measured over a set of representative files, achieves a performance that is as good as the best of the competitive state-of-the-art data compression programs. This set-up of our final design meets the memory requirement, the

speed requirement and the weak performance requirement.

Set-up **Snake** is our preferred option. It uses almost 35 MBytes of memory, it achieves a compression speed of about 6.6 kBytes per second. Now, not only its average performance on a set of representative files is better than the performance of competitive programs, but also its performance on each file is at least comparable. This set-up meets the strong performance requirement, but its memory usage is 10 % above the memory requirement, and its compression speed is a third lower than required. We find the overhead on speed and memory complexity of minor importance, because it still has a lower complexity than the best competitive algorithms.

If we review the requirements summarized in Table 2.1 on page 24, then we may conclude that our final design fulfils all main requirements, that we only sacrificed the sequential behaviour (due to the forward decomposition, the encoder of both set-ups requires two passes over the data), and thus that our project is completed successfully.

9.2 INNOVATIONS

Any thesis needs to contain some innovative ideas. The information in this thesis has been presented as a monologue in which many known algorithms and ideas have been interwoven with new results. This section intends to highlight the most important new contributions of the work described in this thesis (other innovations that have not been part of the final design, and that, consequently, have not been described here are listed in Appendix B).

Chapter 2 presents some new schemes for parallel encoding and decoding. Especially the scheme in which the program works out of phase in the different blocks seems promising. The section on random access briefly describes our first investigations into such schemes [17, 66, 67].

Chapter 4, concerning the CTW algorithm, contains several (small) modifications. The set of computations that perform the weighting in each node have been streamlined with respect to previous implementations, resulting at the same time in a more intuitive form [58]. A pruning strategy has been proposed and evaluated, and although it has not been used in the end, it provided not only a strong alternative, but it also gave some valuable insight into the size of the models formed by CTW and the impact of the maximum depth D. This algorithm has not been published before. Also the way in which unique context paths are being treated has been improved. Furthermore, there are some minor modifications, e.g. the estimator, and the implementation of the hash table.

Chapter 5 discusses the forward decomposition. This decomposition has been used and described before, but mostly as a fixed tree. Here equivalent Huffman trees have been successfully used to construct such a decomposition automatically for each source sequence. This two-pass scheme, together with one of the three heuristics to construct the equivalent Huffman trees, have been published in [58].

Chapter 6 is the heart of this thesis and consists of new work only. The ideas behind the switching method are related to the BIPID algorithm [59], but the set-up of the switching method,

the switching algorithm and the snake algorithm are all new innovations and have been published [56, 57]. The analysis of the snake algorithm has not been published before.

Chapter 7 describes a PPM algorithm. The decomposed implementation of PPM is new and unpublished, and also its set-up and use as companion algorithm for CTW is new.

9.3 RECOMMENDATIONS

The interest in lossless data compression is diminishing at the moment. This seems really strange, since there are so many interesting challenges in the field. We see two categories of challenges: improving the performance of the algorithms, and extending their functionality.

On a very basic level there is of course the ongoing pursuit for algorithms that either result in a better performance, or that have a lower complexity than the existing ones. The entropy of English is estimated at around 1.2 bits per symbol[1]. Currently the best data compression programs might achieve a compression rate of close to 1.6 bits per symbol. Closing this gap is getting less important from a data reduction point of view (because only an additional reduction of at most 25 % seems theoretically possible), but it is still important for other applications, like data mining. If data compression algorithms can improve their performance, then they are better able to extract structural information from the sequences, and consequently better data mining algorithms can be developed.

Performance can be improved in two ways. First, the most straightforward way would be to extend the model class beyond the class of tree sources used by the context-tree weighting algorithm. Some extensions have already been investigated. A first, minor, extension allows don't cares in the contexts [41, 48, 49, 54], giving the program the ability to skip context symbols for some nodes. Such algorithms already have a significant impact on the complexity. Even larger model classes need more complex weighting algorithms [61, 63]. Ultimately one might want to weight over all finite-state machines. The experiments for larger model classes performed in [50, 61, 63] point out an important problem: although it is often not so difficult to find a weighting algorithm for a specific class of models, it is more difficult to find an *efficient* method to compute the weighting formula. In practice, also the complexity of a method determines its success to a great extent. Popular algorithms like Lempel-Ziv algorithms, PPM, and CTW all have a compression time linear with the length of the source sequence. Thus even if a weighting method for an extended class has much better performance, it should also have a low complexity.

In this thesis we chose a different approach to improve the performance. All current data compression algorithms try to find some basic structure in the source sequences. Each of them focuses on a single aspect of the source, and tries to model that particular part of the behaviour

[1] The entropy of English is still a point of discussion. It is even difficult to discuss *the* entropy of English since different types of texts, and texts by different authors will in general have a different entropy. Still it is an interesting question. Shannon [38] and Cover and King [15] let a person guess the next letter in the English text, "Jefferson the Virginian" by Dumas Malone. The subjects only had to guess the next letter or a space, the case of the letters and the punctuation symbols had been omitted. Consequently, these guesses were used to estimate the entropy. Shannon estimated from his experiments that the entropy is between 0.6 and 1.3 bits per symbol, but the lower bound is considered less relevant. Cover and Kind estimate the entropy at around 1.25 bits per symbol.

of the source, but it is clear that none of these algorithms is able to find the true model behind the source. Instead of trying to model more aspects of the source by extending the model classes, one can combine multiple source coding algorithms, such that the combined algorithm benefits from two different models for the behaviour of the source. This idea is supported by the results in this thesis. The biggest improvement in performance achieved by any of the methods presented in this thesis is obtained by the snake algorithm in combination with a companion algorithm, while the increase in complexity is still manageable. Also the results of the Tag-CTW algorithm [6] support this idea. In principle, the Tag-CTW algorithm also consists of two algorithms running "in parallel". The tagger inserts some meta-structural information in the context for CTW by means of the tags. This is an interesting option: one can think of a system with two algorithms. One algorithm tries to find a global structure in the source sequence and it uses this information to steer a second, more locally operating, algorithm, e.g. some version of the CTW algorithm, that actually codes the source sequence. In this respect the compression algorithm that uses context-free grammars [28, 29] to describe the source sequence could be a starting point.

Besides improving the performance (and reducing the complexity), the usage of data compression algorithms can also be enhanced by adding more functionality, and the feature that is probably being missed most is random access. So far, the attempts to achieve random access in data compression fall into one of two categories. On one hand practical methods have been developed (e.g. [20, 70]), but these lack the underlying theoretical foundation, so it is difficult to judge their performance. In general the loss in compression performance caused by introducing random access appears to be quite high, compared to similar schemes without random access. On the other hand there are some mathematical methods [17, 66, 67] that greatly limit the added redundancy, but it has turned out that it is almost unfeasible to implement these methods in practice. Furthermore, because there is no real theoretical basis yet for random access, it is not known whether the redundancy bounds achieved by these schemes are optimal or not. Still, adding random access in a structured way would open many new possibilities for data compression. A second property that is very desirable is parallel processing. This is not only important for a hardware implementation, but can become more important for software implementations too. In Section 2.4.1 some first ideas are briefly described, but especially for adaptive data compression algorithms this is still an open problem.

With so many interesting and important challenges in the data compression field, it is a pity that most work is now concentrated on straightforward applications and minor modifications of existing algorithms, instead of working on more fundamental problems. Obviously, tackling the underlying problems and finding a satisfying, useful, answer will take a lot more time and effort but in the end it will not only be more rewarding, it will turn out to be a dire necessity.

ACKNOWLEDGEMENTS

FIRST, I would like to thank prof. Stevens and prof. Schalkwijk for giving me the opportunity and the support to work on this thesis. Prof. Smeets and prof. de With were so kind to join the reading committee. They all gave me helpful suggestions for improving my work.

I owe some special thanks to Frans Willems, my coach and co-promotor. From our cooperation I learned almost everything I know about data compression, and more. With great pleasure I think back to the numerous discussion sessions in front of his white-board, during which we invented, analysed, evaluated, often rejected and then re-invented, our algorithms and ideas.

With Tjalling Tjalkens and Jan Åberg I had many pleasant technical discussions, most of them related to my work. During my "extended" stay at the university I was so lucky to have two long-time room-mates, Mortaza Shoae Bargh and Yuanqing Guo, who brightened my days. Harry Creemers, Rian van Gaalen, Erik Meeuwissen, Jaap Bastiaans, Ali Nowbakht-Irani, Vladimir Balakirsky, and Eduward Tangdiongga are just some of my colleagues who made working at the university a pleasure.

Finally, I would like to express my gratitude to my parents, Sep and Mieke, for their patient support, my brother Frank for his help with the computer systems, and my sister Marlène for encouraging me to finish my work

They, my friends, and my colleagues from the TUE and from Philips made this a wonderful time for me. Thanks everybody.

A

SELECTING, WEIGHTING AND SWITCHING

A.1 INTRODUCTION

SUPPOSE a source sequence x_1^N was generated by one of two sources, S_1 or S_2. Source S_1 assigns probability $P(x_1^N|S_1)$ to this sequence with corresponding code word $c_1(x_1^N)$, and source S_2 probability $P(x_1^N|S_2)$ and code word $c_2(x_1^N)$. Now, we want a good coding distribution for the source sequence, independently of which of the two sources actually generated the sequence. There are two common approaches, selecting and weighting, while switching might provide a third alternative.

A.2 SELECTING

The encoder can simply compute both $P(x_1^N|S_1)$ and $P(x_1^N|S_2)$, and **select** the source that results in the highest probability. Specifying the selected source to the decoder costs one bit, and consequently, this selecting scheme achieves a coding probability of

$$P_m(x_1^N) = \frac{1}{2}\max(P(x_1^N|S_1),\, P(x_1^N|S_2)), \tag{A.1}$$

on the sequence x_1^N. Suppose model S^* (S_1 or S_2) achieves the highest probability. Then the individual redundancy equals:

$$\rho_m(x_1^N) = l(x_1^N) - \log_2 \frac{1}{P(x_1^N|S^*)} = \log_2 \frac{1}{P_m(x_1^N)} - \log_2 \frac{1}{P(x_1^N|S^*)}$$

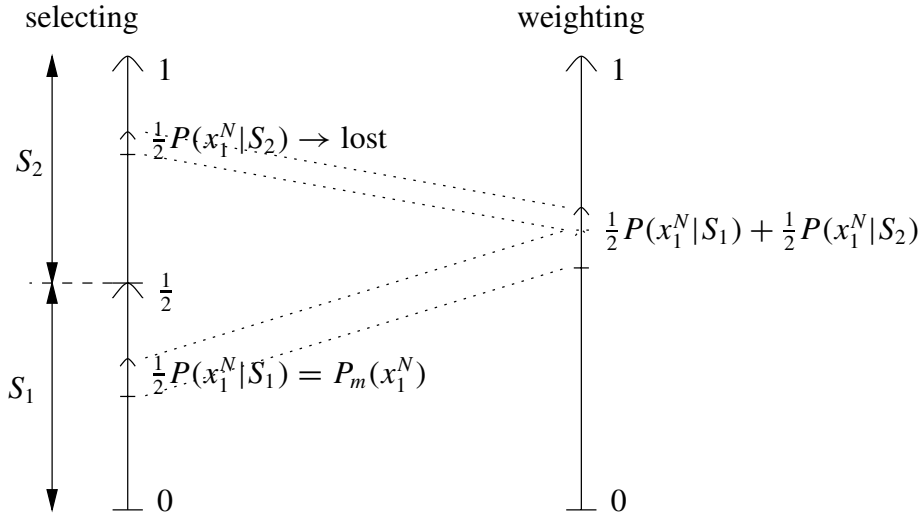


Figure A.1: The difference between selecting and weighting.

$$= \log_2 \frac{P(x_1^N|S^*)}{\frac{1}{2}P(x_1^N|S^*)} = 1, \tag{A.2}$$

if we omit the coding redundancy. But this is not optimal. Say that we use a zero to select source S_1 and a one for source S_2. Suppose that for a particular source sequence $P(x_1^N|S_1) > P(x_1^N|S_2)$ holds, then the encoder will encode this source sequence by a zero followed by $c_1(x_1^N)$. But a one followed by $c_2(x_1^N)$ is also a valid code word that will be correctly decoded to x_1^N, but this code word will never be used. Thus the selecting scheme reserves for each sequence two code words, of which one will never be used, and the code space it occupies is wasted. On the left hand side of Figure A.1 we tried to visualize this. Here we have ordered all code words lexicographical, and placed them in the interval $[0, 1)$ (the same as is done in Section 1.5.5 concerning the arithmetic coder). The code words starting with a zero are the ones that select source S_1, and they occupy the lower half of the interval, while the code words that select source S_2 occupy the upper half. The selecting scheme will reserve two parts of code space for each source sequence, of which one will be lost (one of size $\frac{1}{2}P(x_1^N|S_2)$ for the source sequence mentioned above). An other disadvantage of a selecting scheme is that it has to examine the entire source sequence first, before it can select a source and encode the sequence.

A.3 WEIGHTING

The weighted distribution is defined as:

$$P_w(x_1^n) = \frac{1}{2}P(x_1^n|S_1) + \frac{1}{2}P(x_1^n|S_2), \tag{A.3}$$

for all $n = 1, 2, \ldots, N$. It is a valid coding distribution in which no code space is wasted, since $\sum_{x^n \in \mathcal{X}^n} P_w(x_1^n) = 1$ for all n. Clearly, $P_w(x_1^N) \geq P_m(x_1^N)$. The right hand side of Figure A.1 shows

that the weighted distribution combines the two intervals, instead of selecting one of them. The individual coding redundancy can be upper bounded by,

$$\begin{aligned}\rho_w(x_1^N) &= l(x_1^N) - \log_2 \frac{1}{P(x_1^N|S^*)} = \log_2 \frac{1}{P_w(x_1^N)} - \log_2 \frac{1}{P(x_1^N|S^*)} \\ &= \log_2 \frac{2}{P(x_1^n|S_1) + P(x_1^n|S_2)} - \log_2 \frac{1}{P(x_1^N|S^*)} \\ &\leq \log_2 \frac{2}{P(x_1^n|S^*)} - \log_2 \frac{1}{P(x_1^N|S^*)} = 1, \end{aligned} \tag{A.4}$$

again omitting the coding redundancy. Comparing (A.4) and (A.2) shows that the upper bound on the individual redundancy is the same, thus on some sequences the schemes will perform identical (if one of the sources assigns a zero probability to the sequence), while on other sequences weighting performs better (e.g., if both sources assign the same probability to the source sequence, its redundancy will be zero).

A second advantage of weighting is that it can be used sequentially, since

$$P_w(x_1^n) = P_w(x_1^n, X_{n+1} = 0) + P_w(x_1^n, X_{n+1} = 1)$$

holds[1], for all n. The sequential coding works as follows. Suppose the first symbols x_1^n have already been coded. The encoder first computes $P_w(x_1^{n+1})$, and the conditional probability of the next symbol can then be computed by applying Bayes' rule,

$$P_w(x_{n+1}|x_1^n) = \frac{P_w(x_1^{n+1})}{P_w(x_1^n)} \tag{A.5}$$

$$\begin{aligned} &= \frac{\frac{1}{2}P(x_1^{n+1}|S_1) + \frac{1}{2}P(x_1^{n+1}|S_2)}{\frac{1}{2}P(x_1^n|S_1) + \frac{1}{2}P(x_1^n|S_2)} \\ &= \frac{P(x_1^n|S_1)}{P(x_1^n|S_1) + P(x_1^n|S_2)} P(x_{n+1}|x_1^n, S_1) + \frac{P(x_1^n|S_2)}{P(x_1^n|S_1) + P(x_1^n|S_2)} P(x_{n+1}|x_1^n, S_2). \end{aligned} \tag{A.6}$$

Although (A.5) will be used to compute the conditional probability, (A.6) gives a much better insight. The estimate of each source is weighted with a factor indicating how well that source estimated the previous n symbols: the weight $\frac{P(x_1^n|S_1)}{P(x_1^n|S_1)+P(x_1^n|S_2)}$, is precisely the fraction of the weighted probability contributed by source S_1.

A.4 SWITCHING

In Chapter 6 the switching algorithm has been presented. How does this scheme fit in between selecting and weighting? Switching is based on a weighting technique. It can, compared to

[1]This does **not** hold for the selecting scheme, because if one source results in the highest probability for $x_1^n 0$ and the other one for $x_1^n 1$, then $P_m(x_1^n 0) + P_m(x_1^n 1) > P_m(x_1^n)$. Therefore, the selecting scheme cannot be used sequentially in its current form.

weighting, only redistribute the code space between the possible source sequences: some sequences will get a higher probability, at the cost of other sequences. The switching algorithm is specifically designed for dynamic sources, modelled as switching sources: while generating the source sequence, the source actually switched from one structure to the other. This improved behaviour for such switching sources is obtained at the cost of stationary sources. As a result the upper bound on the individual redundancy, is much higher: $\rho_{sw}(x_1^N) \leq \frac{1}{2}\log_2 N + 1$ if source S_1 achieves the highest probability, and $\rho_{sw}(x_1^N) \leq \frac{3}{2}\log_2 N + 1$ in case of source S_2 (see Chapter 6), omitting the coding redundancy in both cases.

A.5 DISCUSSION

How can these three methods be used in a more general setting, e.g. for compressing sequences generated by tree sources? Selecting, or maximizing, can be used in every node of a context tree to find the best model[2]. There is a two-pass data compression algorithm based directly on this technique, called the context-tree maximizing algorithm, and it is very briefly described in Appendix B. If we weight in every node of the context-tree instead of maximize, we get the context-tree weighting algorithm used in this thesis. The advantages are a better performance (up to one bit per node), and a sequential, or one-pass, program. Theoretically, one could also plug-in switching in every node of the context tree. This will result in a poor performance, because the additional redundancy introduced by the switching algorithm has to be compensated for by an improved performance. Achieving the necessary high gains in performance on a node-level will be almost impossible, therefore we decided to implement switching only at the highest hierarchical level: between the two source coding algorithms. But a closer investigation of the dynamics of a single node in the context-tree weighting algorithm reveals some more possibilities.

Figure A.2 shows four lines. The three lines marked "Memoryless", "Memory $D = 1$", and "Weighted" have been generated by simulations. We used a simple tree source, consisting of one node and two leaves with $\theta_0 = 0.6$ and $\theta_1 = 0.8$ to generate source sequences of 100 symbols. We first assumed that the sequence was generated by a memoryless source and used a single KT-estimator to estimate the probability. Next, we computed the redundancy of this estimate compared to the actual model (with depth $D = 1$). More precisely, we repeated this over 10,000 runs, to obtain the "true" averaged redundancy behaviour of this estimator. This is plotted by the line "Memoryless". We ran the same experiment again, but now assuming that we knew that the tree source had two leaves, such that we had to use two KT-estimators. The resulting individual redundancy, averaged over 10,000 runs, is plotted by the line "Memory $D = 1$". Both redundancy curves start with one bit redundancy, which accounts for specifying the known tree structure: the model cost. From these two redundancy curves we can easily compute the redundancy behaviour of a half-half weighting between the memoryless model and the model with depth $D = 1$, as is performed by CTW. This is indicated by the curve "Weighted". At first the estimate of the memoryless model will dominate the "weighted probability", and finally after 63 symbols, the redundancy of the model with memory becomes lower than the redundancy of the memoryless

[2]The best model is the model that minimizes the description length.

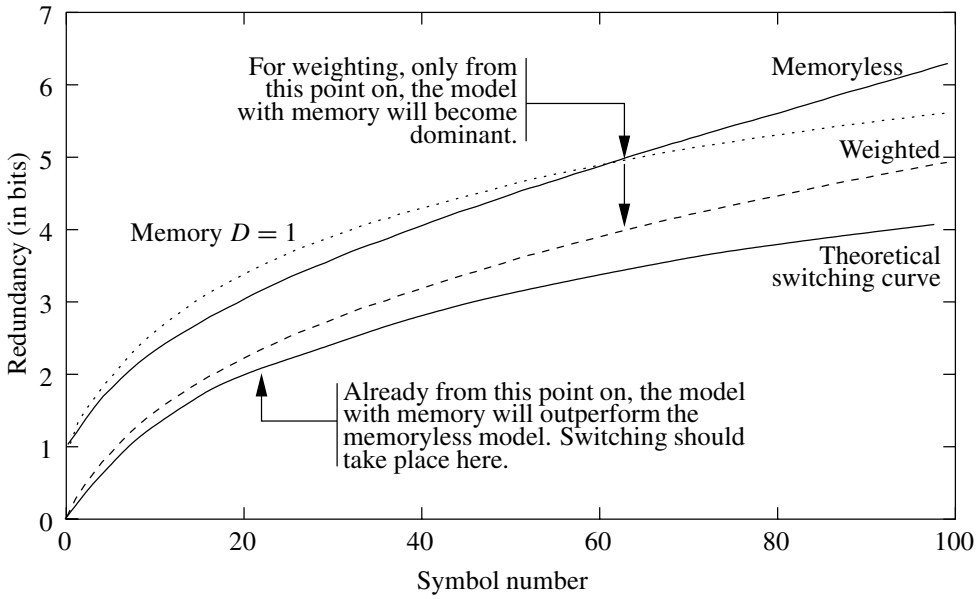


Figure A.2: An example of the average behaviour in a node.

model, and from this point on, the model with memory will start to dominate. The "Weighted" redundancy will converge to the redundancy of the model with memory, and after a few hundred symbols, the redundancy of the weighted model will be indistinguishable from the model with memory.

Is there room for improvement? The answer is: maybe. Around symbol 22, the slope of the redundancy of the memoryless model and the slope of the redundancy of the model with memory are about equal. Before this point, the redundancy of the memoryless model is not only lower, it also **grows** slower than that of the model with memory. After this point the redundancy of the model with memory grows slower than the memoryless one (although it will take another 40 symbols before the initial losses are cancelled). Suppose, a switching algorithm would know this, and it would start by following the memoryless model, and switch to the model with memory after exactly 22 symbols. It would then achieve the redundancy curve marked "Theoretical switching curve", which is more than one bit below the weighted curve, and the gap between them will increase even further. Of course, the switching algorithm presented in Chapter 6 will never achieve this curve, because it introduces a lot of switching redundancy. But, there might be a "stripped-down" version of the switching algorithm, e.g. one that allows only one single switch, that, with some enhancements, could achieve a redundancy below the weighting algorithm. In a practical sense, such a switching algorithm has to switch exactly at the point in which the difference between the memoryless model and the model with memory is largest (in favour of the memoryless model), while its added redundancy should be less than 1 bit (the ultimate model cost in the case of weighting) plus the height of the maximum gap.

Although the idea presented above might be very tempting, a solution is not so self-evident. One has to realize that in most cases even the "optimal" switch will result in very little gain

compared to weighting. And even if an algorithm can be found that achieves a point between the weighted curve and the theoretical switching curve, one has to realize at what costs it has been achieved. If this gain is obtained at the cost of sequences which should never switch (leaves in the actual model), than the solution will be useless for a context-tree *switching* algorithm, because for such algorithms, the performance in leaves is more important than the performance in nodes (since a tree has more leaves than nodes). Only if both the situation with no switches, and the situation with one switch can be solved satisfactory, then it could be a really interesting construction for a CTS algorithm.

B

OTHER INNOVATIONS

THIS appendix gives a brief overview of some new ideas developed during the PhD-project, that have not been used in the final design and that have not been discussed yet.

B.1 THE CONTEXT-TREE MAXIMIZING ALGORITHM

The Context-Tree Maximizing (CTM) algorithm has already been briefly described in Section 2.4. This two-pass algorithm, has a lower encoder complexity and a much lower decoder complexity than the CTW algorithm, at the cost of a decrease in performance. In the first pass the CTM algorithm will compute the "best" model and it will describe this model for the decoder. In the second pass it will encode the source sequence, given this fixed model.

In the first pass, the CTM algorithm will compute the Minimum Description Length (MDL) model, the model that minimizes the length of describing the model itself plus the length of describing the source sequence given this model. To compute this model, the CTM algorithm will first count the zeros and ones in every node and leaf of the context tree for the entire source sequence. Next, it walks through the context tree depth-first, and in every node and leaf s of the context tree it will compute the estimated probability $P_e^s(x_1^N)$ and the following maximized probability,

$$P_m^s(x_1^N) = \begin{cases} P_e^s(x_1^N), & \text{if } l(s) = D, \\ \frac{1}{2}\max(P_e^s(x_1^N),\, P_m^{0s}(x_1^N) \cdot P_m^{1s}(x_1^N)), & \text{if } l(s) < D. \end{cases} \tag{B.1}$$

Furthermore, it will mark nodes s in which $P_e^s(x_1^N)$ is the maximum as a "leaf", and the other nodes as a "node" of the MDL model. The tree described by the marks "node" and "leaf" is the desired MDL model. The CTM algorithm concludes the first pass by describing this MDL model for the decoder. It is encoded by walking depth-first through the MDL model, writing a

"0" for every unvisited node and a "1" for every leaf (except for leaves at depth D). The cost for describing the model, one bit per node and leaf of the MDL model, are accounted for in the computation of the MDL model by the factor a half in (B.1).

Now, for the second pass, the problem is reduced to the simple problem of a source with a known tree structure (see also Section 4.2.2). The encoder will use counters in each leaf of the MDL model and the KT-estimator to estimate the probability of the source sequence.

The CTM algorithm described here will not perform very well. The model cost of the CTM algorithm are exactly one bit per node or leaf, while the model cost of the CTW algorithm are *at most* one bit per node or leaf (see also the discussion in Appendix A). This difference in model cost is the cause of the significantly higher redundancy of the CTM algorithm. Two important improvements of the CTM algorithm were found during the PhD-project, the *model description on the fly* and the *improved model description* (see [51, 52, 53]), both aimed at reducing the model cost of CTM. Model description on the fly is a technique that mixes the description of the model and the description of the source sequence. It tries to delay the description of the MDL model as long as possible, and it only describes a part of the MDL model when that specific part of the MDL model is essential for the decoder to decode the next source symbol correctly. In the end the encoder will have described only those parts of the MDL model that are actually being used, e.g. omitting all unvisited leaves, consequently reducing the model cost (especially for non-binary source alphabets) enormously. The improved model description uses a memoryless estimator to encode the model description. If the part of the MDL model that is being described has significantly more leaves than nodes (or vice versa) then this too will result in a considerable gain.

An other result obtained during the PhD-project is an upper bound on the maximum size of the MDL model. One might expect that the maximum number of leaves in a MDL model grows logarithmically with the length of the sequence. This appears not to be the case. If we are allowed to freely distribute the counts in an infinitely deep context tree, then with some N counts, the maximum number of leaves of the largest MDL model converges from below to $\frac{1}{3}N$ for $N \to \infty$. We were not able to find a source sequence that actually achieves this bound, but the worst case source sequence that we could find resulted in a MDL model with roughly $\frac{1}{4}N$ leaves. These results have not been published.

Finally we were able to apply a version of the CTM algorithm to machine learning [50].

B.2 EXTENDED MODEL CLASSES FOR CTW

The CTW algorithm assumes that the source belongs to the class of tree sources. Already in [61, 63] weighting algorithms for larger model classes have been presented. In these papers the CTW algorithm for tree sources is called class IV, the least complex class of algorithms. Class III covers a more complex class of sources: it weights over any permutation of the context symbols (see also the structure on the left hand side of Figure B.1). In practice its "context tree" is constructed as pairs of splits, first one of the yet unused context positions between 1 and D is chosen (e.g. the root node in Figure B.1 splits in context positions one and two), and in the next node the context symbol at that position is used to split the contexts. Since this class results

in a too high computational complexity, a model class between class III and class IV has been investigated: we extended the class of tree sources by allowing "don't cares" in the context (see [54]). In this class context symbols can be skipped for determining the parameter. The resulting context tree structure has been drawn in the middle of Figure B.1, an asterisk denotes a don't care. For this class we defined the following weighted probability,

$$P_w^s(x_1^n) = \begin{cases} P_e^s(x_1^n) & \text{if } l(s) = D, \\ \alpha P_e^s(x_1^n) + (1-\alpha)\left\{(1-\beta)P_w^{*s}(x_1^n) + \beta P_w^{0s}(x_1^n) P_w^{1s}(x_1^n)\right\} & \\ \qquad \text{if } l(s) = 0 \text{ or if } 0 < l(s) < D, \text{ and last symbol } s \text{ not an } *, & \\ (1-\beta)P_w^{*s}(x_1^n) + \beta P_w^{0s}(x_1^n) P_w^{1s}(x_1^n) & \\ \qquad \text{if } 0 < l(s) < D, \text{ and last symbol } s \text{ is an } *. \end{cases} \tag{B.2}$$

The second equation is a weighting over all three options: it is a leaf, a node or this context symbol can be skipped. The third equation is used for nodes that follow a don't care branch. After a don't care it is useless to create a leaf, because then this leaf could already have been created in the parent node.

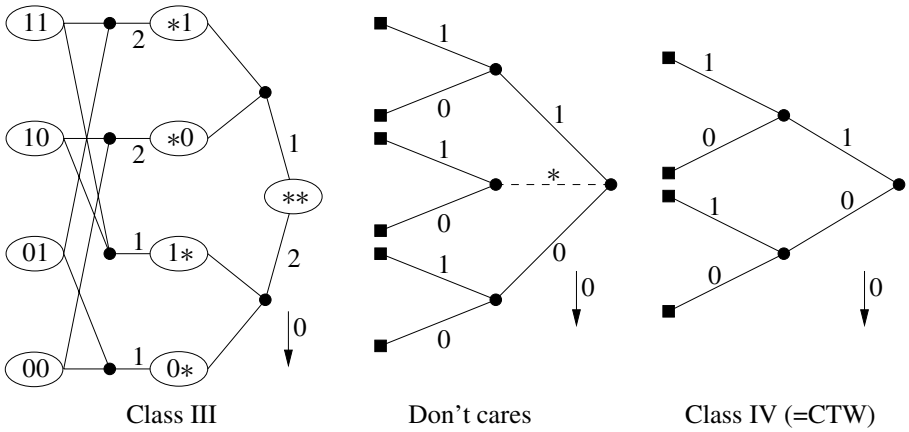


Figure B.1: Comparing the extended model class with don't cares to other classes

The complexity of this extended model class is about 2^{D-1} updates per symbol, which is lower than the class III algorithm, but still too high to use in practice with a reasonable depth D. This shows an interesting property, also mentioned in the conclusions (Chapter 9): extending the model class is only useful if one can find an efficient weighting technique for that class.

Some success has been obtained with this extended model class by combining it with a backward decomposition [48, 49]. The backward decomposition decomposes the context symbols in smaller subsymbols. E.g., we have used a decomposition that decomposes the context symbols in two parts: first a global description of the symbol (vowel, consonant, number, etc.), followed by the exact specification within its group ("e", "m", "2", etc.). By applying don't cares the CTW algorithm has the possibility to use only the global description of a context symbol, and skip the detailed information. A partial description of the context symbols has as advantage that counts from different contexts will be combined, possibly resulting in more accurate estimates.

Although a small performance improvement has been achieved, in [48, 49] it is concluded that the gain in performance does not justify the increase in computational complexity.

B.3 THE CONTEXT-TREE BRANCH-WEIGHTING ALGORITHM

The context-tree branch-weighting algorithm [55] is a derivative of the CTW algorithm. The CTW algorithm weights over nodes: a node in the context tree is either considered to be a leaf of the actual model, or a node together with all its children. But it is also possible to do the opposite and weight over branches: a branch is either used or it is not. This has as a consequence that a node should act as a leaf for some context symbols and as a node for others. The advantage is clear, all symbols that use this node as leaf share the same parameter. This reduces the number of parameters, and could increase the accuracy of the parameters.

There is a difficulty with the branch-weighting algorithm. Consider ternary context symbols. In a node s the algorithm could use either 0, 1, 2 or all 3 branches. This results in 8 possible combinations, and for 7 of these combinations different probabilities $P_e^s(.)$ have to be computed, depending on which symbols contribute to the $P_e^s(.)$. For larger alphabets this exact computation of the branch-weighting algorithm will become even more computationally complex. Therefore, we use an approximation for the different $P_e^s(.)$ terms, which will not only result in a computational complexity equal to that of the CTW algorithm, but also in a beautiful weighting formula.

The estimated probability in a node s can be written as:

$$P_e^s(x_1^n) \stackrel{\triangle}{=} \prod_{t \in \mathcal{X}} P_e^{(t)s}(x_1^n), \tag{B.3}$$

in which $P_e^{(t)s}$ is the probability of all symbols following context ts, estimated with the estimator in node s each time they occurred. Thus if a new symbol x_{n+1} has context ts probability $P_e^{(t)s}$ will be updated with the estimate of the estimator in node s: $P_e^{(t)s}(x_1^{n+1}) = P_e^{(t)s}(x_1^n) P_e^s(X_{n+1} = x_{n+1}|x_1^n)$. The other probabilities $P_e^{(y)s}$ with $y \neq t$ will not change. We will use the probability $P_e^{(t)s}$ as an approximation for the contribution of the single branch corresponding to context ts to the total estimated probability $P_e^s(.)$. Thus, if more branches of a node s are not used and should contribute to the estimated probability, then the approximated $P_e^s(.)$ will be computed by simply multiplying the approximations for each branch, e.g.,

$$P_e^{\{0,1\}s}(x_1^n) \approx P_e^{(0)s}(x_1^n) P_e^{(1)s}(x_1^n), \tag{B.4}$$

$$P_e^{T's}(x_1^n) \approx \prod_{t \in T'} P_e^{(t)s}(x_1^n), \tag{B.5}$$

in which $T' \subseteq \mathcal{X}$. With these approximations the branch-weighting algorithm can be formulated as:

$$P_{bw}^s(x_1^n) = \prod_{t \in \mathcal{X}} \left(\frac{1}{2} P_e^{(t)s}(x_1^n) + \frac{1}{2} P_{bw}^{ts}(x_1^n) \right), \quad \text{for } l(s) < D. \tag{B.6}$$

This probability distribution is a good coding distribution since $P_{bw}^s(\emptyset) = 1$, and $P_{bw}^s(x_1^n) = \sum_{t \in \mathcal{X}} P_{bw}^s(X_{n+1} = t, x_1^n)$. The equation in (B.6) can be explained in a very intuitive way. For

every *branch t* the algorithm weights between two probabilities: the probability in case the actual model has a leaf for that context and the probability in case it has a branch for that context. In the first case the estimated probability of the symbols with context *ts*, estimated with the estimator in this node *s* ($P_e^{(t)s}$) is used, and in the latter case the weighted probability of the child node (P_{bw}^{ts}) is used. Notice that the branch-weighting algorithm, (B.6), is the *perfect* dual of the regular context-tree weighting algorithm:

$$P_w^s(x_1^n) = \frac{1}{2}\prod_{t\in\mathcal{X}} P_e^{(t)s}(x_1^n) + \frac{1}{2}\prod_{t\in\mathcal{X}} P_w^{ts}(x_1^n), \quad \text{for } l(s) < D. \tag{B.7}$$

The computational and memory complexity of the branch-weighting algorithm is the same as the complexity of CTW. Experiments show that the performance of the branch-weighting algorithm (with different weights) is competitive with that of the CTW algorithm!

B.4 AN INCOMPRESSIBLE SEQUENCE

We also investigated compressibility in a more theoretical way. Ziv and Lempel [73] introduced a measure for the compressibility of sequences. We showed that some sequences that are "incompressible" by this definition, meaning that they are "incompressible by any finite-state information-lossless encoder" (and as a result by Lempel-Ziv'78 algorithms), can be compressed by a variation of the CTW algorithm [3]. This shows that the concept of compressibility is somewhat counter-intuitive.

REFERENCES

[1] J. Åberg. *A Universal Source Coding Perspective on PPM*. PhD thesis, Department of Information Technology, Lund University, October 1999.

[2] J. Åberg and Yu.M. Shtarkov. Text compression by context tree weighting. In *Proc. Data Compression Conference*, pages 377–386, Snowbird, Utah, March 25 - 27 1997.

[3] J. Åberg, P. Volf, and F. Willems. Compressing an incompressible sequence. In *IEEE Int. Symp. on Information Theory*, page 134, MIT, Cambridge, Massachusetts, August 16 - 21 1998.

[4] M. Abramowitz and I.A. Stegun, editors. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, number 55 in Applied Mathematics Series. U.S. Department of Commerce, National Bureau of Standards, June 1964.

[5] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proc. Data Compression Conference*, pages 201–210, Snowbird, Utah, March 25 - 27 1997.

[6] J. Bastiaans. Tag-CTW. Masters thesis, Eindhoven University of Technology, August 1998. In Dutch.

[7] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[8] L.C. Benschop. *Lossless Data Compression in VLSI*. PhD thesis, Department of Electrical Engineering, Eindhoven University of Technology, 1997.

[9] C. Bloom. PPMZ-2 v0.7. http://www.cbloom.com/, 1999.

[10] S. Bunton. *On-Line Stochastic Processes in Data Compression*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1996.

[11] S. Bunton. A percolating state selector for suffix-tree context models. In *Proc. Data Compression Conference*, pages 32–41, Snowbird, Utah, March 25 - 27 1997.

[12] J.G. Cleary, W.J. Teahan, and I.H. Witten. Unbounded length contexts for PPM. In *Proc. Data Compression Conference*, pages 52–61, Snowbird, Utah, March 28-30 1995.

[13] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications*, 32(4):396–402, 1984.

[14] T.M. Cover. Enumerative source encoding. *IEEE Trans. on Inform. Theory*, 19(1):73–77, 1973.

[15] T.M. Cover and R.C. King. A convergent gambling estimate of the entropy of English. *IEEE Trans. on Inform. Theory*, 24(4):413–421, July 1978.

[16] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, 1991.

[17] G. Enzner. Random access coding. Trainee report, Eindhoven University of Technology, April 1999.

[18] P.G. Howard. The design and analysis of efficient lossless data compression systems. Report CS-93-28, Department of Computer Science, Brown University, Providence, Rhode Island, 1993.

[19] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40:1098–1101, 1952.

[20] K. Iwata, T. Uyematsu, and E. Okamoto. Proposal of partially decodable Ziv-Lempel code. In *IEEE Int. Symp. on Information Theory*, page 22, Whistler, British Columbia, Canada, September 17-22 1995.

[21] F. Jelinek. Buffer overflow in variable length coding of fixed rate sources. *IEEE Trans. on Inform. Theory*, 14(3):490–501, 1968.

[22] G.G. Langdon jr. A note on the Ziv-Lempel model for compressing individual sequences. *IEEE Trans. on Inform. Theory*, 29(2):284–287, 1983.

[23] G.G. Langdon jr. and J.J. Rissanen. Method and means for carry-over control in the high order to low order pairwise combining of digits of a decodable set of relatively shifted finite number strings. US Patent, 4463342, 1984.

[24] G.O.H. Katona and T.O.H. Nemetz. Huffman codes and self-information. *IEEE Trans. on Inform. Theory*, 22(3):337–340, 1976.

[25] R. Koster. Compressie van teksten met behulp van het contextweegalgorithme. Stage verslag, Eindhoven University of Technology, June 1994. In Dutch.

[26] R.E. Krichevsky and V.K. Trofimov. The performance of universal encoding. *IEEE Trans. on Inform. Theory*, 27(2):199–207, 1981.

[27] A. Moffat. Implementing the PPM data compression scheme. *IEEE Trans. on Communications*, 38(11):1917–1921, 1990.

[28] C.G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, Department of Computer Science, University of Waikato, 1996.

[29] C.G. Nevill-Manning, C.G. Witten, and D.L. Maulsby. Compression by induction of hierarchical grammars. In *Proc. Data Compression Conference*, pages 244 – 253, Snowbird, Utah, March 29 - 31 1994.

[30] R.C. Pasco. *Source Coding Algorithms for Fast Data Compression*. PhD thesis, Department of Electrical Engineering, Stanford University, May 1976.

[31] J.J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of research and development*, 20:198–203, May 1976.

[32] J.J. Rissanen. Universal coding, information, prediction and estimation. *IEEE Trans. on Inform. Theory*, 30(4):629–636, 1984.

[33] J.J. Rissanen and G.G. Langdon jr. Arithmetic coding. *IBM Journal of research and development*, 23:149–162, March 1979.

[34] J.J. Rissanen and G.G. Langdon jr. Universal modeling and coding. *IEEE Trans. on Inform. Theory*, 27(1):12–23, 1981.

[35] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Trans. on Inform. Theory*, 25(6):672–675, 1979.

[36] J.P.M. Schalkwijk. An algorithm for source coding. *IEEE Trans. on Inform. Theory*, 18(3):395–398, 1972.

[37] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

[38] C.E. Shannon. Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:50–64, January 1951.

[39] Yu.M. Shtarkov. Switching discrete sources and its universal encoding. *Problems of Information Transmission*, 28(3):95–111, July - Sept. 1992.

[40] J.A. Storer. *Data Compression: Methods and Theory*. Number 13 in Principles of computer science series. Computer science press, Rockville, Maryland, 1988.

[41] J. Suzuki. A CTW scheme for some FSM models. In *IEEE Int. Symp. on Information Theory*, page 389, Whistler, British Columbia, Canada, September 17-22 1995.

[42] M. Taylor. rk. http://malcolmt.tripod.com/rk.html, 2000.

[43] W.J. Teahan and J.G. Cleary. Models of English text. In *Proc. Data Compression Conference*, pages 12–21, Snowbird, Utah, USA, March 25 - 27 1997.

[44] W.J. Teahan and J.G. Cleary. Tag based models of English text. In *Proc. Data Compression Conference*, pages 43–52, Snowbird, Utah, USA, March 30 - April 1 1998.

[45] Tj.J. Tjalkens. A new hashing technique for CTW. Personal communication, 1998.

[46] Tj.J. Tjalkens, Yu.M. Shtarkov, and F.M.J. Willems. Context tree weighting: Multi-alphabet sources. In *Symp. on Inform. Theory in the Benelux*, volume 14, pages 128–135, Veldhoven, The Netherlands, May 17-18 1993.

[47] Tj.J. Tjalkens, F.M.J. Willems, and Yu.M. Shtarkov. Multi-alphabet universal coding using a binary decomposition context tree weighting algorithm. In *Symp. on Inform. Theory in the Benelux*, volume 15, pages 259–265, Louvain-la-Neuve, Belgium, May 30-31 1994.

[48] P.A.J. Volf. Text compression methods based on context weighting. Technical report, Stan Ackermans Institute, Eindhoven University of Technology, June 1996.

[49] P.A.J. Volf. Context-tree weighting for text-sources. In *IEEE Int. Symp. on Information Theory*, page 64, Ulm, Germany, June 29 - July 4 1997.

[50] P.A.J. Volf and F.M.J. Willems. Context maximizing: Finding MDL decision trees. In *Symp. on Inform. Theory in the Benelux*, volume 15, pages 192–200, Louvain-la-Neuve, Belgium, May 30 - 31 1994.

[51] P.A.J. Volf and F.M.J. Willems. The context tree maximizing method : Some ideas. In *EIDMA Winter Meeting on Coding Theory, Information Theory and Cryptology*, page 27, Veldhoven, the Netherlands, December 1994.

[52] P.A.J. Volf and F.M.J. Willems. On the context tree maximizing algorithm. In *IEEE Int. Symp. on Information Theory*, page 20, Whistler, British Columbia, Canada, September 17 - 22 1995.

[53] P.A.J. Volf and F.M.J. Willems. A study of the context tree maximizing method. In *Symp. on Inform. Theory in the Benelux*, volume 16, pages 3–9, Nieuwerkerk a/d IJsel, The Netherlands, May 18 - 19 1995.

[54] P.A.J. Volf and F.M.J. Willems. Context-tree weighting for extended tree sources. In *Symp. on Inform. Theory in the Benelux*, volume 17, pages 95–101, Enschede, The Netherlands, May 30-31 1996.

[55] P.A.J. Volf and F.M.J. Willems. A context-tree branch-weighting algorithm. In *Symp. on Inform. Theory in the Benelux*, volume 18, pages 115–122, Veldhoven, The Netherlands, May 15 - 16 1997.

[56] P.A.J. Volf and F.M.J. Willems. Switching between two universal source coding algorithms. In *Proc. Data Compression Conference*, pages 491–500, Snowbird, Utah, March 30 - April 1 1998.

[57] P.A.J. Volf and F.M.J. Willems. The switching method: Elaborations. In *Symp. on Inform. Theory in the Benelux*, volume 19, pages 13–20, Veldhoven, The Netherlands, May 28-29 1998.

[58] P.A.J. Volf, F.M.J. Willems, and Tj.J. Tjalkens. Complexity reducing techniques for the CTW algorithm. In *Symp. on Inform. Theory in the Benelux*, volume 20, pages 25–32, Haasrode, Belgium, May 27-28 1999.

[59] F.M.J. Willems. Coding for a binary independent piecewise-identically-distributed source. *IEEE Trans. on Inform. Theory*, 42(6):2210–2217, 1996.

[60] F.M.J. Willems. Efficient computation of the multi-alphabet estimator. Personal communication, 2000.

[61] F.M.J. Willems, Yu.M. Shtarkov, and Tj.J. Tjalkens. Context weighting: General finite context sources. In *Symp. on Inform. Theory in the Benelux*, volume 14, pages 120–127, Veldhoven, The Netherlands, May 17-18 1993.

[62] F.M.J. Willems, Yu.M. Shtarkov, and Tj.J. Tjalkens. The context-tree weighting method: Basic properties. *IEEE Trans. on Inform. Theory*, 41(3):653–664, 1995.

[63] F.M.J. Willems, Yu.M. Shtarkov, and Tj.J. Tjalkens. Context weighting for general finite-context sources. *IEEE Trans. on Inform. Theory*, 42(5):1514–1520, 1996.

[64] F.M.J. Willems and Tj.J. Tjalkens. Complexity reduction of the context-tree weighting algorithm: A study for KPN research. Technical Report EIDMA-RS.97.01, Euler Institute for Discrete Mathematics and its Applications, Eindhoven University of Technology, 1997.

[65] F.M.J. Willems and Tj.J. Tjalkens. Complexity reduction of the context-tree weighting method. In *Symp. on Inform. Theory in the Benelux*, volume 18, pages 123–130, Veldhoven, The Netherlands, May 15-16 1997.

[66] F.M.J. Willems, Tj.J. Tjalkens, and P.A.J. Volf. On random-access data compaction. In *IEEE Information Theory and Communications Workshop*, page 97, Kruger National Park, South Africa, June 20-25 1999.

[67] F.M.J. Willems, Tj.J. Tjalkens, and P.A.J. Volf. On random-access data compaction. In I. Althöfer, N. Cai, G. Dueck, L. Khachatrian, M.S. Pinsker, A. Sárközy, I. Wegener, and Z. Zhang, editors, *Numbers, Information and Complexity*, pages 413–420, Boston, 2000. Kluwer Academic Publishers.

[68] F.M.J. Willems and P.A.J. Volf. Reducing model cost by weighted symbol decomposition. In *IEEE Int. Symp. on Information Theory*, page 338, Washington, D.C., United States, June 24-29 2001.

[69] R.N. Williams. *Adaptive Data Compression*. Kluwer Academic Publishers, Boston, 1991.

[70] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.

[71] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[72] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inform. Theory*, 23(3):337–343, 1977.

[73] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Inform. Theory*, 24(5):530–536, 1978.

INDEX

CURRICULUM VITAE

PAUL Volf was born on September 12, 1971 in Eindhoven, the Netherlands. In September 1989 he started as an Electrical Engineering student, specialization Information Technology, at the Eindhoven University of Technology. He performed his graduation work in the Information and Communication Theory group. This work was concerned with applying data compression algorithms to data mining problems. He received his master's degree on April 14, 1994.

In June 1994 he started on a project for the post-graduate designers study "Information and Communication Technology" at the Stan Ackermans Institute. His project dealt with the design and implementation of new data compression programs. He worked on the project in the Information and Communication Theory group. He successfully completed the study in June 1996. He continued this project as Ph.D. student from October 1996 to the end of 1999 in the same group. This thesis is the result of that work.

Since January 1, 2000, he is employed at Philips Semiconductors, first as DfM engineer, later as product engineer.

Stellingen

behorende bij het proefontwerp

Weighting Techniques in Data Compression: Theory and Algorithms

van Paul Volf.

1. "All current data compression algorithms try to find some basic structure in the source sequences. Each of them focuses on a single aspect of the source, and tries to model that particular part of the behaviour of the source, but it is clear that none of these algorithms is able to find the true model behind the source. Instead of trying to model more aspects of the source by extending the model classes, one can combine multiple source coding algorithms, such that the combined algorithm benefits from two different models for the behaviour of the source."
   Dit proefontwerp, hoofdstuk 9, verwijzend naar hoofdstuk 6

2. Tag-CTW increases the source sequence length, without increasing the code word length, which effectively proves that an alternative system to combine two different modelling algorithms – one main compression algorithm, controlled or steered by an other modelling algorithm – is also a viable concept.
   Dit proefontwerp, hoofdstuk 2, sectie 2.4.3

3. In [1] sequences are constructed that can be compressed with a rate < 1, by CTW, but which are "incompressible" according to the compressibility measure defined in [2]. This is fundamentally different from the opposite: showing that there are sequences that can be compressed according to some compressibility measure but not by CTW — a practical universal source code — simply confirms that CTW has been designed for the restricted class of tree sources.
   Dit proefontwerp, appendix B, sectie B.4
   [1] J. Åberg, P. Volf, and F. Willems. Compressing an incompressible sequence. *IEEE Int. Symp. Inform. Theory*, p. 134, MIT, Cambridge, MA., August 1998.
   [2] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, vol. 24, pp. 530–536, 1978.

4. The weighted symbol decomposition CTW [1], aka the K'NEX CTW, is an almost ideal solution for extending CTW towards multi-alphabets, however, its performance is not significantly better than existing CTW solutions for multi-alphabets, because it achieves its efficiency by using nested binary KT-estimators to mimic true multi-alphabet estimators, resulting in additional parameter redundancy.
   [1] F.M.J. Willems and P.A.J. Volf. Reducing model cost by weighted symbol decomposition. *IEEE Int. Symp. Inform. Theory*, p. 338, Washington, D.C., June 2001.

5. “Strong opens defects can cause a circuit to malfunction, but even weak open defects can cause it to function poorly. [...] [We] verified the correlation between weak opens and delay faults. From this analysis, we can conclude that in modern deep-submicron technologies, the incidence of weak opens is high enough to require delay-fault testing.”
   R. Rodríguez, P. Volf and J. Pineda. Resistance characterization for weak open defects, *IEEE Design & Test of Computers*, vol. 19, no. 5, pp. 18-26, 2002.

6. Meettijden van wafers op testers gedragen zich volgens een normaal-verdeling met eventueel een additionele staart naar langere meettijden, die een indicatie is voor problemen met de opstelling. De maat “mediaan(X)/gemiddelde(X)” voor zo’n tijdsdistributie X is een snelle afschatting voor de efficiëntie van de opstelling.

7. Voor de doorstroming is het beter, als automobilisten bij een rood verkeerslicht op drie meter afstand van hun voorganger stoppen.

8. “Tool-commonalities” en het identificeren van systematische uitval op wafermaps, zijn slechts twee voorbeelden van alledaagse problemen in de semiconductor industrie, die met datacompressie algoritmes aangepakt kunnen worden.

9. In navolging van concerten en computerspelletjes, verdient het aanbeveling dat ook proefschriften eindigen met een toegift.

10. Het gezegde “een plaatje zegt meer dan 1000 woorden” gaat niet op voor veel automatisch gegenereerde engineering rapporten.

11. Als men overal cameratoezicht gaat toepassen, dan moet iedereen ook portretrecht krijgen.