LOSSLESS COMPRESSION AND ALPHABET SIZE

by

DANIEL A. NAGY

A thesis submitted to the Department of Mathematics and Statistics

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

January, 2006

Abstract

Lossless data compression through exploiting redundancy in a sequence of symbols is a well-studied field in computer science and information theory. One way to achieve compression is to statistically model the data and estimate model parameters. In practice, most general purpose data compression algorithms model the data as stationary sequences of 8-bit symbols. While this model fits very well the currently used computer architectures and the vast majority of information representation standards, other models may have both computational and information theoretic merits in being more efficient in implementation or fitting some data closer. In addition, compression algorithms based on the 8 bit symbol model perform very poorly on data represented by binary sequences not aligned with byte boundaries either because the fixed symbol length is not a multiple of 8 bits (e.g. DNA sequences) or because the symbols of the source are encoded into bit sequences of variable length.

Throughout this thesis, we assume that the source alphabet consists of blocks of equal size of elementary symbols (typically bits), and address the impact of this block size on lossless compression algorithms in general and in the context of so-called block-sorting compression algorithms in particular. These algorithms are quite popular both in theory and in practice and are the subjects of intensive research with many interesting results in recent years.

We show that compression on the bit level is tolerant to sources that are not aligned to byte boundaries, while performing reasonably well for byte-aligned sources.

More generally, we prove upper bounds on the average codelength redundancy resulting from not taking block alignment into account when modeling the source for compression. An extensive information-theoretic analysis and experimental evaluation is presented. We address both the information theoretic and algorithmic issues of doing lossless compression on small alphabets with an emphasis on the binary case.

We also develop and analyze a simple and efficient algorithm that can be used for block sorting, where the length of the input alphabet symbols (as measured in bits) is a parameter. The asymptotic performance of our algorithm achieves (up to a constant factor) the lower bound on running time determined by the computational complexity of the problem of suffix sorting on such alphabets, while its simplicity matches that of the simplest alternatives. The data structure that it is based on — the suffix tree — has many uses in string processing in general and lossless compression in particular beyond block sorting.

Acknowledgements

I would like to acknowledge the help and support of my supervisor Tamás Linder. I am also indebted to András György for his help in the information theoretic analysis in this thesis.

Contents

List of Tables

List of Figures

Notation

A	Discrete finite alphabet. A *set* of finitely many symbols.
A^n	Set of all possible strings of n symbols from A.
A^*	Set of all possible finite strings from A.
a, b, a_i, b_i	Binary symbols, elements of $\{0, 1\}$.
a_i^j	Binary string. Shorthand for $a_i, a_{i+1}, \ldots, a_j$.
a_0^∞	Infinite binary sequence: $a_0, a_1, a_2, \ldots$
$D(X \| Y)$	Relative entropy between random variables X and Y. See (2.18).
$\bar{D}(X_0^\infty \| Y_0^\infty)$	Relative entropy rate between random processes X_0^∞ and Y_0^∞. See (2.22).
$H(X)$	Entropy of random variable X. See (2.2).
$H(X \mid Y)$	Conditional entropy of r. v. X given r. v. Y. See (2.3).
$\bar{H}(X_0^\infty)$	Entropy rate of X_0^∞. See (2.5).
$L(X)$	Average encoded length of X. See (2.10).
$\bar{L}(X_0^\infty)$	Rate of encoding, average per-symbol code length. See (2.12).
$\mathbb{N}$	The set of non-negative integers.
$\mathbb{N}^+$	The set of positive integers.
$O(f(n))$	$c_n = O(f(n))$ iff $\limsup\limits_{n\to\infty} \dfrac{c_n}{f(n)} \leq const.$
$o(f(n))$	$c_n = o(f(n))$ iff $\lim\limits_{n\to\infty} \dfrac{c_n}{f(n)} = 0.$
$P_X(x)$	Probability of $X = x$.
$P_{X\mid Y}(x\mid y)$	Conditional probability of $X = x$ given $Y = y$.

$R(X)$	Redundancy of coding X. $R(X) = L(X) - H(X)$. See (2.14).
$\bar{R}(X_0^\infty)$	Redundancy rate of X_0^∞. See (2.17).
$\mathbb{R}$	The set of real numbers.
$\mathbb{R}^+$	The set of positive real numbers.
X, Y	Discrete random variables or their distributions (typically over A).
X_i^j	Discrete random vector. Shorthand for $X_i, X_{i+1}, \ldots, X_j$.
X_0^∞	Discrete random process. Shorthand for $X_0, X_1, X_2, \ldots$.
x, y, x_i, y_i	Symbols from A.
x_i^j	String of symbols from A. Shorthand for $x_i, x_{i+1}, \ldots, x_j$.
$\mathbf{x}, \mathbf{y}$	Finite strings of symbols from A. $\mathbf{x} = x_0^{\lvert\mathbf{x}\rvert-1}$
$\lvert\mathbf{x}\rvert$	Length of (number of symbols in) $\mathbf{x}$.
$\mathbf{xy}$	Concatenation of strings $\mathbf{x}$ and $\mathbf{y}$.
$\overrightarrow{\mathbf{x}\vert\mathbf{y}}$	Arc in a directed graph pointing from vertex $\mathbf{x}$ to vertex $\mathbf{y}$.
x_0^∞	Infinite sequence of symbols from A. Shorthand for $x_0, x_1, x_2, \ldots$.

Chapter 1

Introduction

Since the advent of the telegraph, the nineteenth century forerunner of digital electronic communication, it has always been a design goal to communicate as much information as possible over a given period of time. If the signaling rate attains the fastest possible over the communication channel in question, further improvement can be achieved by carefully choosing the code used to represent the information communicated. One of the earliest examples of a code designed with conciseness in mind that has been (and still is) in wide use comes from the telegraph: the Morse code. In this code, shorter codes correspond to more frequent letters (in the English language), thus making the average length of messages (in English) shorter.

Designing codes minimizing the length of representation is the subject of *data compression*. As long as we require identical reconstruction of the original message from the concise representation, we speak of *lossless* data compression. Another field of data compression is *lossy* data compression, where the reconstructed data is not identical to the original but retains its salient characteristics. While the most dramatic reductions in transmission costs are achieved by lossy compression, it should be noted that more often than not, lossy compression schemes depend on an underlying lossless compression algorithm. In the vast majority of cases, the information deemed unimportant is first removed from the data which is then losslessly compressed.

Reducing the time (and thus the costs) of communication is not the only application of data compression, however. Concise representation can also reduce the costs of storing information by reducing the amount of symbols that need to be recorded. In both of these applications, efficient compression entails significant economic benefits. For historical reasons, data compression is usually considered a branch of the broader field of communication theory. Since the foundations of information theory have been laid out by Claude Elwood Shannon in his historic paper from 1948 [38], data compression often uses Shannon's terminology for communication. For example, the information that needs to be compressed is said to come from a *source* and compression itself is referred to as *source coding*.

In contemporary digital communication (and recording) the vast majority of information is represented by a sequence of ones and zeros (binary digits, *bits*) at every stage of processing except, maybe, input and output. Hence, the symbols of the alphabets representing the information are either bits or blocks thereof. While most results in information theory apply to other alphabets as well, the binary case remains the most important and most studied one.

Also for historic reasons, the most common block size is eight bits (an *octet* or *byte*), but not all digital information is represented as a sequence of bytes. Therefore, it is interesting to analyze what happens if the assumption of byte alignment does not hold and investigate the consequences of dropping this assumption.

This thesis is about the impact of block size on the performance of block-sorting compression. The criteria by which we evaluate performance are the redundancy of the source coding, which is the excess of the length of the encoded output over the theoretical minimum, and the time and memory requirements under the standard (RAM) model of computation.

1.1 Contributions

The contributions of this thesis are as follows:

- We prove an explicit expression for the divergence rate between a block Markov source (see Definition 2.4.2) and a higher-order Markov model. This expression is a lower bound on the redundancy rate of a compression scheme approximating the block Markov source with a higher order Markov model. In particular, we show that the limit of this divergence rate is zero as the memory of the model approaches infinity, and characterize the rate of convergence.

- For source codes that are strongly universal (see Definition 2.5.2) for higher order Markov sources, we prove an upper bound on their actual redundancy rate, when applied to block Markov sources. This result implies that strongly universal codes for higher order Markov sources are universal for block Markov sources as well.

- We generalize a popular data structure used in block sorting and other lossless compression algorithms, the suffix tree (see Definition 2.10.2), for alphabets consisting of blocks of equal size of elementary symbols. We prove that this generalized data structure (the *block suffix tree*, Definition 5.2.1) can be constructed using time and memory proportional to the block size and the length of the input.

- For the case when the source alphabet consists of blocks of bits, we present an efficient representation and develop an algorithm (Algorithm 5.4.1) for the construction of the block suffix tree using this representation, which compares favorably with available alternatives in terms of simplicity and performance.

1.2 Thesis Overview

The thesis is organized in the following manner:

In Chapter 2, we review the theoretical background in information theory and computer science for our work.

First, we set up the information theoretic framework for discussing the problem of source coding. Without proofs, we present fundamental results from the literature concerning the limits of compression. Then we discuss the evaluation of source codes. We review the most important source models and introduce the concept of *universal source coding*. The information theoretic part of the chapter is concluded by the description of *arithmetic coding*.

We continue by introducing *block sorting compression* and the *Burrows-Wheeler transformation*. We briefly discuss different variations on this scheme that have been published since the original paper by Burrows and Wheeler. The rest of the chapter is dedicated to results concerning the *suffix tree*, which is a potent tool for block sorting and has also other applications in string processing and source coding.

In Chapter 3, an experimental evaluation of a simple binary block-sorting compressor is presented. We describe in detail the experimental setup: the binary block sorting and the subsequent modeling and encoding.

In the experiments themselves, we compare our compression algorithm to two industry standard lossless compressors: `bzip2`, which is a block sorting compressor operating on bytes and `gzip`, which is an LZ-77 implementation, also on bytes. We use the *Calgary compression corpus* and its zero-order Huffman-encoded version to compare the compression algorithms, as this way we can compare sources with the same entropy one of which is aligned to byte boundaries while the other is not. We present the experimental results in a table.

We observed that for sources not aligned to byte boundaries, our compression algorithm consistently outperformed those operating on bytes, while providing reasonable compression for byte-aligned inputs.

In Chapter 4, we analyze the above scenario from an information theoretic point of view. For the sake of simplicity, we assume the source to be *block Markov* and the source coder's model to be *higher order Markov* on elementary symbols.

First, we find the source model that best approximates the source in the *divergence rate* sense. The resulting minimum divergence rate has the operational meaning of being the coding redundancy if a block-N Markov source is encoded using the best source model from the set of order-m Markov models. We show that as the model's memory increases, the divergence rate converges to zero exponentially fast.

Then we look at how *strongly universal codes* for higher order Markov sources perform when applied to block Markov sources with an unknown distribution. We prove an upper bound on the redundancy of such codes and discuss the implications of our result.

In Chapter 5, we generalize the suffix tree on a fixed alphabet to a *block suffix tree* on uniform blocks of arbitrary length. We prove that the block suffix tree can be constructed in linear time in an on-line fashion using a generalized version of Ukkonen's algorithm and that it constitutes a representation of the suffix tree on the larger alphabet of blocks.

We generalize the binary suffix tree representation used for the experiments in Chapter 3 and derive a binary block suffix tree representation and describe an algorithm to construct it, which compares favorably in terms of simplicity and performance with available alternatives for suffix sorting on blocks of symbols.

In Chapter 6, we summarize and interpret our results in terms of design criteria for lossless compression applications, revisit the experimental results in Chapter 3 with the hindsight of our later results and provide directions for possible future work on the subject.

Chapter 2

Literature Review and Theoretical Background

In this chapter, the theoretical background of our work is reviewed. Basic concepts and fundamental results from the literature are presented without proofs.

2.1 Entropy, Entropy Rate and the Limits of Compression

In information theory, sources of information are modeled with (discrete) random processes. In the discrete case, this means a sequence of random variables $X_0, X_1, X_2, \ldots, X_i, \ldots$ (denoted henceforth by X_0^∞) over some finite discrete alphabet A. Let $P_X(x)$ denote the probability that the discrete random variable X equals x, and let X_i^j and x_i^j, where $i \leq j$, denote $X_i X_{i+1} \ldots X_j$ and $x_i x_{i+1} \ldots x_j$, respectively. Random processes must satisfy the *consistency conditions* for any positive integer $n \in \mathbb{N}^+$ and $x_0^{n-1} \in A^n$:

$$P_{X_0^{n-1}}(x_0^{n-1}) = \sum_{y \in A} P_{X_0^n}(x_0^{n-1} y). \tag{2.1}$$

The Appendix of [11] provides a concise summary of process concepts.

Formally, a discrete random variable is a mapping $X : \Omega \to A$ of the probability space Ω onto the alphabet A. Similarly, a discrete random process $X_0^\infty : \Omega \to A^\infty$ is a mapping of Ω onto infinite sequences of symbols from A, satisfying (2.1). The distributions of random variables and processes are due to the probability measure P defined over a sigma-algebra on the subsets of Ω.

Following Shannon [38], we introduce the notion of information-theoretic entropy:

Definition 2.1.1 (Entropy) *For a discrete random variable X we define its entropy as the negative expected value of the base 2 logarithm of the probability of the outcome:*

$$H(X) = E\left[-\log P_X(X)\right] = -\sum_{x\in A} P_X(x)\log P_X(x) \tag{2.2}$$

where log *stands for base 2 logarithm.*

Definition 2.1.2 (Conditional Entropy) *For two random variables X and Y, their conditional entropy is defined as the entropies of the conditional distributions of X weighed by the probabilities of the conditions:*

$$H(X \mid Y) = \sum_{y\in A} P_Y(y)H(X \mid Y = y) \tag{2.3}$$

where

$$H(X \mid Y = y) = -\sum_{x\in A} P_{X|Y}(x \mid y)\log P_{X|Y}(x \mid y) \tag{2.4}$$

Entropy and conditional entropy as defined above are expressed in *bit*s.

Definition 2.1.3 (Entropy Rate) *For the random process $X_0^\infty = X_0, X_1, \ldots,$ its entropy rate is defined as*

$$\bar{H}(X_0^\infty) = \lim_{n\to\infty}\frac{1}{n}H\left(X_0^{n-1}\right) \tag{2.5}$$

if the limit exists.

Entropy rate is expressed in *bits/symbol.*

An alternative definition of the entropy rate is the limit of a symbol's entropy conditioned on all past symbols:

$$\bar{H}'(X_0^\infty) = \lim_{n\to\infty} H\left(X_n \mid X_0^{n-1}\right) \tag{2.6}$$

It can be shown [8, Theorem 4.2.1] that the limits in the two definitions in (2.5) and (2.6) exist and are equal for stationary processes (see Definition 2.3.1). In particular, if X_0^∞ is a stationary order-n Markov process (see Section 2.3.3), where the distribution of a symbol depends only on the previous n symbols, the entropy rate is as follows:

$$\bar{H}(X_0^\infty) = H\left(X_i \mid X_{i-n}^{i-1}\right) \tag{2.7}$$

for any $i > n$.

The binary representation of sequences of source symbols is called a *code*. Formally, we treat the code as a sequence of functions

$$f_n : A^n \to \{0,1\}^*, \tag{2.8}$$

where n is a non-negative integer and $\{0,1\}^*$ denotes the set of finite binary strings. A set of binary strings is said to be *uniquely decodable* iff all finite concatenations of its elements result in different binary strings. The code f_n is uniquely decodable, if the set $\{f_n\left(x_0^{n-1}\right) : x_0^{n-1} \in A^n\}$ is uniquely decodable.

A binary string a_0^n is said to be a *prefix* of another binary string b_0^m iff $n \leq m$ and $a_0^n = b_0^n$. Similarly, c_0^n is a *suffix* of b_0^m iff $n \leq m$ and $b_{m-n}^m = c_0^n$.

A special class of uniquely decodable codes are the *prefix codes* (a.k.a. instantaneous codes), where no codeword is a prefix of another codeword and so the code-words can be decoded immediately as their last bits become available. Interestingly, the code length achievable by prefix codes is not worse than that of the best uniquely decodable code.

It can be shown that the average code length in a uniquely decodable representation of a random variable cannot be shorter than the entropy of the random variable. The proof hinges on the Kraft inequality, which states that all uniquely decodable codes satisfy:

$$\sum_{x \in A} 2^{-\ell(x)} \leq 1 \tag{2.9}$$

where $\ell(x)$ denotes the length of the codeword corresponding to symbol x. The minimization of average code length

$$L(X) = \sum_{x \in A} P_X(x)\ell(x) \tag{2.10}$$

constrained by (2.9) yields

$$L(X) \geq H(X). \tag{2.11}$$

Also, for any code where (2.9) is an equality, there exists a distribution for which the code is optimal, that is $L(X) = H(X)$. This distribution corresponding to the code is $P_X(x) = 2^{-\ell(x)}$ for all $x \in A$.

An immediate implication for independent, identically distributed (I.I.D.) sources is that the average per symbol code length of uniquely decodable representations is limited from below by the entropy rate of the source. This result can be easily generalized to other classes of sources.

When dealing with sequences of source symbols, instead of the average code length, we often use the average per-symbol code length, or the *rate of encoding*:

Definition 2.1.4 (Rate of Encoding) *Given the sequence of uniquely decodable codes*

$\{f_n\}$

$$\bar{L}(X_0^\infty) = \lim_{n\to\infty} n^{-1} \sum_{x_0^{n-1}\in A^n} P_{X_0^{n-1}}(x_0^{n-1}) \left|f_n(x_0^{n-1})\right| \tag{2.12}$$

where $\left|f_n(x_0^{n-1})\right|$ *denotes the length of* $f_n(x_0^{n-1})$ *and* $P_{X_0^{n-1}}(x_0^{n-1})$ *denotes the probability that source* X_0^∞ *emits* x_0^{n-1} *as the first* n *symbols of its output. Here we assume that the limit exists.*

For a given distribution of X, it is always possible to construct a code for which $L(X) \leq H(X) + 1$. Shannon already provides an example of such a code, which he attributes to Fano. This code is the so-called *Shannon-Fano code* [8], where

$$\ell(x) = \lceil -\log P_X(x) \rceil \text{ for every } x \in A. \tag{2.13}$$

For a random process, if the individual symbols are encoded in an instantaneous fashion, meaning that each symbol can be decoded as soon as the last bit in its representation is received, we get an overhead of at most one bit per symbol. However, by not requiring instantaneous decoding, we can "spread" this overhead over several symbols by grouping them together for coding. There are several ways of achieving this. For instance, the Shannon-Fano code for blocks of N symbols yields $\ell(x_0^{N-1}) = \lceil -\log P_{X_0^{N-1}}(x_0^{N-1}) \rceil$ for every block $x_0^{N-1} \in A^N$, providing for an overhead of at most 1 bit for N symbols.

In theory, a method called *arithmetic coding* [25] (see also [26] and Section 2.6) can asymptotically always achieve compression to the entropy rate, given accurate estimates of the conditional probabilities of the coming symbols. Arithmetic coding is another example of spreading the coding overhead by encoding several symbols together. In practice, however, the computational costs in buffer size and numerical precision, both of which can grow without bounds, often render direct arithmetic coding unfeasible. In these cases other methods should be devised for compression that are computationally feasible and efficient. Yet, as illustrated by various codes only slightly exceeding the entropy rate of the source, the lion's share of the problem in data compression is the accurate estimation of the symbol probabilities — the statistical modeling of the source. The next section deals with measuring the accuracy of the modeling.

2.2 Redundancy, Relative Entropy, Divergence Rate

The amount by which a code length exceeds the entropy (rate) of a source is called the redundancy (rate) of that code with respect to the source. Equation (2.14) defines the redundancy of encoding $f_1(X)$ of a single random variable X:

$$R(X) = L(X) - H(X) = \sum_{x\in A} P_X(x)\ell(x) - \sum_{x\in A} P_X(x)\log\frac{1}{P_X(x)} = \sum_{x\in A} P_X(x)\log\frac{P_X(x)}{2^{-\ell(x)}}. \tag{2.14}$$

where $\ell(x) = |f_1(x)|$

Similarly, one can define the redundancy of encoding a random vector of length n using f_n:

Definition 2.2.1 (Redundancy)

$$R(X_0^{n-1}) = \sum_{x_0^{n-1}\in A^n} P_{X_0^{n-1}}(x_0^{n-1})\log\frac{P_{X_0^{n-1}}(x_0^{n-1})}{2^{-|f_n(x_0^{n-1})|}}. \tag{2.15}$$

Definition 2.2.2 (Redundancy Rate) *The redundancy rate of some code f for some source X_0^∞ is the limit of the per-symbol redundancies of the elements of sequence f_n as n approaches infinity. This equals the difference between the rate of encoding and* *the entropy rate of the source (when the limits exist):*

$$\begin{aligned}\bar{R}(X_0^\infty) &= \lim_{n\to\infty}\frac{1}{n}\left(\sum_{x_0^{n-1}\in A^n} P_{X_0^{n-1}}(x_0^{n-1})\ell(x_0^{n-1}) - H\left(X_0^{n-1}\right)\right) && (2.16)\\ &= \bar{L}(X_0^\infty) - \bar{H}(X_0^\infty). && (2.17)\end{aligned}$$

Definition 2.2.3 (Relative Entropy) *For two random variables X and Y, one can define*

$$D\left(X \parallel Y\right) = \sum_{x\in A} P_X(x)\log\frac{P_X(x)}{P_Y(x)}. \qquad (2.18)$$

This quantity is known as the *relative entropy* of X and Y or the *Kullback-Leibler distance* between their distributions after the authors of the original paper [24] where it was first introduced. The latter is denoted by $D(P_X\|P_Y)$ and used interchangeably with $D(X\|Y)$ throughout the thesis. It is not a proper distance measure between the distributions of X and Y (e.g., it is not symmetric), but it retains a very important property of distance measures: It is non-negative and equals zero iff the two distributions are equal.

If f_1 satisfies Fano's inequality (2.9) with an equality, then (2.14) and (2.18) become the same expression by setting the distribution of Y so that $P_Y(x) = 2^{-\ell(x)}$. In the general case, let us define the distribution of Y so that $P_Y(x) = 2^{-\ell(x)}/Q$ where $Q = \sum_{x\in A} 2^{-\ell(x)}$. Clearly,

$$D\left(X \parallel Y\right) = R(X) + \log Q \le R(X), \qquad (2.19)$$

since $Q \le 1$ according to (2.9).

On the other hand, the Shannon-Fano code lengths $l'(x)$ for the probability mass function $P_Y(x)$ satisfy both

$$l'(x) = \lceil -\log P_Y(x)\rceil \le -\log P_Y(x) + 1. \qquad (2.20)$$

and $l'(x) \leq \ell(x)$ for every symbol $x \in A$. For this new, improved code, the redundancy $R'(X)$ satisfies

$$D\left(X \parallel Y\right) \leq R'(X) \leq D\left(X \parallel Y\right) + 1. \tag{2.21}$$

Thus, we are justified to conclude that good codes can be well modeled by probability distributions and the redundancy of such codes is essentially given by the relative entropy of the source and model distributions.

Similarly to the entropy rate as in (2.5), one can define the *divergence rate* (a.k.a. *relative entropy rate*) of two sources as follows:

Definition 2.2.4 (Divergence Rate)

$$\bar{D}\left(X_0^\infty \parallel Y_0^\infty\right) = \lim_{n\to\infty} \frac{1}{n} D\left(X_0^{n-1} \parallel Y_0^{n-1}\right) \tag{2.22}$$

whenever the limit exists.

Redundancy is a natural measure of quality for codes designed for data compression. As explained above, the relative entropy of the source distribution and the *coding distribution* (the one with the PMF $P_Y(x)$) is a good estimate of redundancy for efficient codes that cannot be improved by assigning shorter codes to some symbols without a penalty on others. If coding is performed on n-long blocks of symbols, the per-symbol redundancy can be estimated to the precision of n^{-1} using the relative entropy of the source and coding distributions.

2.3 Some Stationary Source Models

A source is called *stationary* if the joint distributions of the source symbols do not change if all the indices are incremented by the same value. The formal definition is the following:

Definition 2.3.1 (Stationary Process) *The process X_0^∞ is stationary if for any $i, s, n \in \mathbb{N}$ and $x_0^{n-1} \in A^n$*

$$P_{X_i^{i+n-1}}(x_0^{n-1}) = P_{X_{i+s}^{i+s+n-1}}(x_0^{n-1}). \tag{2.23}$$

A stationary source X_0^∞ is called *ergodic*, iff it cannot be represented as a mixture of two different stationary processes. Any stationary process can be represented as a unique mixture of ergodic processes (by the so-called *ergodic decomposition theorem*). Ergodic processes with a finite expected value are the broadest class for which the strong law of large numbers holds [8, Section 15.7], as expressed by *Birkhoff's Ergodic Theorem*:

$$\frac{1}{n}\sum_{i=0}^{n-1} X_i \to E[X_0]\,(= E[X_1]\ldots) \text{ with probability 1 as } n \to \infty \tag{2.24}$$

2.3.1 Independent, Identically Distributed (I.I.D.) Source

This is the most important class of memoryless (independent) sources. In this model, every symbol emitted by the source comes from the same distribution. This is the only stationary memoryless source model.

Definition 2.3.2 (I.I.D. Source) *A random sequence of symbols X_0^∞ from A is an* I.I.D. source *if $X_0, X_1, \ldots$ are independent and for any $i, j \in \mathbb{N}$*

$$P_{X_i}(x) = P_{X_j}(x) \tag{2.25}$$

holds for any $x \in A$.

The entropy rate of an I.I.D. source is simply the entropy of the symbol distribution:

$$\bar{H}\left(X_0^\infty\right) = H\left(X_i\right) \tag{2.26}$$

for any index i.

2.3.2 Markov Source

In this model the source has a memory of one symbol. The successive symbols in this source form an irreducible, homogeneous, stationary Markov chain.

Definition 2.3.3 (Markov Source) *A random process X_0^∞ is a* stationary Markov source *iff*

$$\begin{aligned}
P_{X_i|X_0^{i-1}}\left(x_i \mid x_0^{i-1}\right) &= P_{X_i|X_{i-1}}\left(x_i \mid x_{i-1}\right) &(2.27)\\
P_{X_{i+1}|X_i}\left(x_1 \mid x_0\right) &= P_{X_{j+1}|X_j}\left(x_1 \mid x_0\right) &(2.28)\\
P_{X_0}\left(x_0\right) &= P_{X_n}\left(x_0\right) &(2.29)
\end{aligned}$$

for any $i, j, n \in \mathbb{N}$ and $x_0, x_1, x_i \in A$.

It can be show that the entropy rate of a Markov source is given by

$$\bar{H}\left(X_0^\infty\right) = H\left(X_{i+1} \mid X_i\right) = \sum_{x \in A} P_{X_i}(x)\, H\left(X_{i+1} \mid X_i = x\right). \tag{2.30}$$

The I.I.D. source is a special case of the Markov source.

2.3.3 Order-n Markov Source

This stationary source has a finite memory. An order-n Markov source has a memory of n symbols. In this case, the random vectors $(X_i, X_{i+1}, \ldots, X_{i+n-1})$ where $i = 0, 1, \ldots,$ form a stationary Markov chain. As a consequence, for any $i, j \in \mathbb{N}$, $y \in A$ and $x_0^{n-1} \in A^n$

$$P_{X_{i+n}|X_i^{i+n-1}}\left(y \mid x_0^{n-1}\right) = P_{X_{j+n}|X_j^{j+n-1}}\left(y \mid x_0^{n-1}\right). \tag{2.31}$$

Stationary order-n Markov sources are fully characterized by the conditional distributions of a symbol given all possible vectors of n preceding symbols. Alternatively, they can be characterized by the joint distribution of $n+1$ subsequent symbols.

The entropy rate of a higher order Markov source is the linear combination of the entropies of the transitional distributions weighed by the stationary probabilities of the source states, as defined in (2.7):

$$\bar{H}(X_0^\infty) = \sum_{x_0^{n-1} \in A^n} P_{X_0^{n-1}}(x_0^{n-1}) H(X_n | x_0^{n-1}) \tag{2.32}$$

I.I.D. and Markov sources are special cases of this model, with order 0 and order 1, respectively.

2.3.4 Tree Source

The tree source is an important special case of higher-order Markov sources. Let $\mathcal{T}$ denote a finite set of strings of symbols from A so that for any sufficiently large value of n for all $x_0^n \in A^{n+1}$ there is exactly one $\mathbf{t}(x_0^n) \in \mathcal{T}$ such that $x_{n-|\mathbf{t}(x_0^n)|+1}^n = \mathbf{t}(x_0^n)$. This implies that if $\mathbf{y} \in \mathcal{T}$ then $y_i^{|\mathbf{y}|-1} \notin \mathcal{T}$ for $i > 0$. That is no element of $\mathcal{T}$ is a suffix of any other element in $\mathcal{T}$.

Definition 2.3.4 (Tree Source) *A random process X_0^∞ is a* tree source *if for any given $z \in A$, $x_0^n, y_0^n \in A^{n+1}$ and $i, j \in \mathbb{N}$, $\mathbf{t}(x_0^n) = \mathbf{t}(y_0^n)$ implies*

$$P_{X_{i+n+1}|X_i^{i+n}}(z|x_0^n) = P_{X_{j+n+1}|X_j^{j+n}}(z|y_0^n). \tag{2.33}$$

In other words, such a source is a special order $n+1$ Markov source, where the transitional probabilities are determined by the immediate past from $\mathcal{T}$.

If the length of the longest string in $\mathcal{T}$ is m, then the tree source is an order m Markov source. Note that since $|\mathcal{T}|$ can be much smaller than $|A|^m$, the tree source may require much fewer parameters for its definition, namely the transition probabilities of a tree machine [40].

Example: Consider a source emitting random words separated by whitespace characters, where the words are drawn from a uniform distribution over a finite dictionary. On the character level, this is a tree source, as its memory is no longer than the longest word in the dictionary plus one symbol. Let us denote the length of the longest word by n. The transitional distributions corresponding to past vectors of $n + 1$ symbols that differ only before the last whitespace are equal. Thus, in this example, $\mathcal{T}$ consists of the union all possible strings of $n + 1$ symbols without whitespace and all possible strings of $n + 1$ or less symbols where only the first symbol is a whitespace.

Examples of data that are well modeled by tree sources include written texts in a natural language. Somewhat surprisingly, the empirical entropy of tree models constructed from natural language texts is smaller if the text is considered in reversed direction. We have not yet seen a satisfactory explanation to this finding, although the observation has already been made in [6].

2.4 Block Sources

2.4.1 Block I.I.D. Sources

Definition 2.4.1 (Block I.I.D. Source) *A sequence of random variables* $X_0, X_1, \ldots$ *is a* Block-N I.I.D. source*, if for some positive integer* N*, the random vectors* $Y_i = (X_{iN}, X_{iN+1}, \ldots, X_{iN+N-1})$ *are independent and identically distributed.*

Obviously, all block-N I.I.D. sources are also block-kN I.I.D. sources for all positive integer values of k. Therefore, all I.I.D. sources are block I.I.D. sources, but the converse is not true.

The entropy rate in the sense of (2.5) for a block-N I.I.D. source X_0^∞ is

$$\bar{H}(X_0^\infty) = \frac{\bar{H}(Y_0^\infty)}{N}. \tag{2.34}$$

2.4.2 Block Markov Sources

Markov models are traditionally the starting point of analyzing processes with memory. Hence, for the analysis of block sources with memory, it is instrumental to have a better understanding of Markov sources over blocks of symbols.

Definition 2.4.2 (Block Markov Source) *The sequence of random variables $X_0, X_1, \ldots$ taking values in a finite alphabet A is a* block-N Markov source*, if for some positive integer N the random sequence $Y_0, Y_1, \ldots$ forms a homogeneous, stationary Markov chain, where $Y_i = X_{iN}^{iN+N-1}$ for all $i \in \mathbb{N}$.*

One can also define higher order block Markov sources. It has to be mentioned, however, that an order m Markov source over blocks of N symbols is also a block-Nm Markov source, so most results carry over directly to higher order block Markov sources. Also, higher order Markov sources are block-Markov sources, but the converse is not true.

2.4.3 Using Block Models for Data Compression

For the purposes of data compression, grouping the symbols of the source into blocks of uniform length may or may not be advantageous. Using the simplest example of modeling an I.I.D. source with a block I.I.D. source, both cases can be illustrated.

In case of optimal coding and an I.I.D. source with a known distribution, the lower bound on the expected code length is the same whether the coding is done on the symbol level or the level of blocks of N symbols, namely the entropy rate of the source. Moreover, one can actually gain from coding blocks rather than individual symbols by spreading the overhead over the blocks as explained in Section 2.1. The average per-symbol code length for optimal codes on N-symbol blocks for an I.I.D. source satisfies

$$\bar{L}(X_0^\infty) \leq \frac{1}{N} H(X_0^{N-1}) + \frac{1}{N}, \tag{2.35}$$

where $\frac{1}{N} H(X_0^{N-1}) = H(X_0) = \bar{H}(X_0^\infty)$. Thus, in this example, the redundancy rate vanishes with $\frac{1}{N}$ as N increases.

However, if the distribution is not known and we resort to estimating it using a finite sample of n symbols coming from the source with the unknown distribution, the picture is quite different.

In the following example, we estimate ("learn") model parameters for two different source models using a finite sample and subsequently code a sequence from the same source that is independent of the "learning" sample. By doing the estimation on blocks rather than individual symbols, one increases the cardinality k of the set over which the distribution needs to be estimated from $k = |A|$ to $k' = |A|^N$ while at the same time decreasing the sample to work from.

Example: Consider a binary I.I.D. (Bernoulli) source. A sample of 16 bits coming from this source is quite usable for estimating the parameter of the underlying Bernoulli distribution. If, however, we group these bits into two bytes, it becomes unfeasible to estimate the distribution of an I.I.D. source on bytes from these two sample values.

Krichevsky shows [22] that for the best estimate (the one for which the bound on redundancy converges to zero fastest as the sample length approaches infinity), the worst case redundancy of a code constructed using this estimate has the following sharp upper bound

$$R(X) \leq \log \frac{\Gamma(n + k/2)\, \Gamma(1/2)}{\Gamma(n + 1/2)\, \Gamma(k/2)} \leq \frac{k-1}{2} \log n - \log \frac{\Gamma(k/2)}{\Gamma(1/2)} + O\left(n^{-1}\right), \tag{2.36}$$

where Γ denotes the usual gamma-function.

By inspecting (2.36), it is immediately obvious that blocking the symbols by N, which results in $k' = k^N$ and $n' = n/N$, affects the expected redundancy adversely, especially when normalized by n for a per-symbol measure. In the above example $k = 2$, $n = 16$ and $N = 8$, yielding $k' = 256$ and $n' = 2$ and an approximately fivefold increase in the redundancy bound according to (2.36). Thus, using a more general class of sources for estimating the model parameters does not come for free. The price we pay is popularly known as data dilution: estimating more model parameters from the same amount of data results in a less accurate estimate.

Depending on the source, it is well possible that for codes constructed by estimating the model parameters on a finite sequence, the redundancy achieved by an I.I.D. source code will be smaller than that by a block code even if the block I.I.D. model models the source more accurately in the asymptotic sense.

2.5 Universal Source Coding

As discussed in Section 2.2, sources with known distributions can be compressed to within 1 bit of their entropy. In many cases, however, very little prior information is available about the data to be compressed and one is compelled to use universal (adaptive) data compression algorithms. Usually, we treat such processes as though they were coming from a reasonably broad class of sources (e.g., one of those introduced in the previous section). Section 5 in [11] gives a detailed overview of the fundamental results on universal source coding.

Definition 2.5.1 (Universal Code) *A sequence $\{f_n\}$ of uniquely decodable codes is called a* universal code *with respect to a family of sources iff for any source X_0^∞ in* *the family*

$$\lim_{n\to\infty}\left(n^{-1}\left[\sum_{x_0^{n-1}\in A^n} P_{X_0^{n-1}}\left(x_0^{n-1}\right)\left|f_n(x_0^{n-1})\right| - H\left(X_0^{n-1}\right)\right]\right) = 0$$

That is the difference between the average encoded length of the source sequences and their entropy (per source symbol) converges to zero as the length of the source sequences approaches infinity.

One may also require that the redundancy rate converge to zero uniformly over each source in the class and each source sequence, leading to the following definition:

Definition 2.5.2 (Strongly Universal Code) *A universal code* $\{f_n\}$ *is* strongly universal *for a class of sources, if for some sequence* c_n *so that* $\lim_{n\to\infty} c_n = 0$ *it satisfies*

$$n^{-1}\sup_{P_{X_0^{n-1}}}\max_{x_0^{n-1}\in A^n}\left[\left|f_n\left(x_0^{n-1}\right)\right| + \log P_{X_0^{n-1}}\left(x_0^{n-1}\right)\right] \le c_n,$$

where the supremum is taken over the marginal distributions of the first n *symbols in all sources within the class.*

In a relative entropy sense, the distribution corresponding to the universal code is close enough to all distributions within the given source model so that the K-L distance (2.18) divided by the length of the source sequence converges to zero, as the latter approaches infinity. In geometric terms, it means that the distribution corresponding to the universal code is near the "center" of the set of probability distributions characterizing the source family.

For classes of sources where the number of *types* increases sub-exponentially (e.g. in a polynomial fashion) with the length of the sequence, it is possible to construct universal codes (for which the divergence rate can be arbitrarily small). Types are equivalence classes of output sequences where appropriately defined empirical distributions are the same within a class. The idea of types is extensively developed in the book on information theory by I. Csiszár and J. Körner [9]. For example, in case of I.I.D. sources two sequences of equal length belong to the same type if the symbol counts are equal for all symbols in the source alphabet. In case of Markov sources, two sequences belong to the same type, if for each symbol the symbol counts following that particular symbol are equal.

For these classes of sources, it is easy to show that universal codes exist, since the information (rate) required to distinguish between the types is a logarithm of a sub-exponential function of n divided by n, which converges to zero. In plain English, the description of the type vanishes compared to the description of the actual sequence within the type. For more general classes of sources, the proof of existence of universal codes can be more involved and actually constructing universal codes for them is even more difficult; see e.g. [35] for an example for a universal code for stationary sources. Universal codes exist for sources as general as the class of piecewise stationary block sources [43, 44, 14].

The simplest example of a class of sources for which universal codes exist is the class of binary I.I.D. sources. A simple universal code would look as follows: first, we encode the length n of the sequence in a code of length $2\lceil\log(n)\rceil - 1$ (see e.g. [15] for such a code) then we encode the number k of ones in the sequence using base 2 binary representation in $\lceil\log n\rceil$ bits and finally the sequence itself as a sequence of Bernoulli random variables with $p = \frac{k}{n}$ with some entropy-achieving code (e.g. a Golomb code of the runs [18, 15]). The redundancy of such a code as defined in (2.17) can be arbitrarily close to zero, since the encoding of n and k vanishes when divided by n as it approaches infinity, and the rest is arbitrarily close to the entropy. It has to be noted that this is by no means the best universal code; there are others for which the redundancy converges to zero faster [23].

When it is difficult to find a well-matching source model for the data that need to be compressed (e.g. text in some unknown language), universal codes are used.

However, always going for the most general universal code can be inefficient, for reasons similar to the case of codes designed using model parameters estimated on a finite sample, as discussed in Section 2.4.3. See [23] for a discussion of the impact of alphabet size on the performance of optimal universal coding of I.I.D. sources.

The most important property of a universal code is the rate of convergence of the worst case redundancy rate to zero, as the length of the encoded sequence increases. Since in practice we always compress finite sequences, the actual redundancy depends to a large extent on this rate.

Let us define the least possible value for worst case redundancy on source sequences of n symbols as follows:

Definition 2.5.3 (Minimax Redundancy) *Given a class of sources,* minimax redundancy R_n^* *for uniquely decodable codes* $f_n : A^n \to \{0,1\}^*$ *with an average code length* $L(X_0^{n-1})$ *is defined as*

$$R_n^* = \min_{f_n} \sup_{X_0^{n-1}} \frac{1}{n} \left(L(X_0^{n-1}) - H(X_0^{n-1}) \right), \tag{2.37}$$

where the supremum is taken over all possible nth order marginal distributions of the first n source symbols in the given source class and the minimum is taken over all uniquely decodable codes.

From [11, Theorem 7.5], for different classes of processes we have the following minimax redundancies:

1. For the class of I.I.D. processes

$$R_n^* = \frac{|A| - 1}{2n} \log n + O(n^{-1}). \tag{2.38}$$

2. For the class of Markov processes

$$R_n^* = \frac{|A|\,(|A|-1)}{2n}\log n + O(n^{-1}). \tag{2.39}$$

3. For the class of mth order Markov processes

$$R_n^* = \frac{|A|^m\,(|A|-1)}{2n}\log n + O(n^{-1}). \tag{2.40}$$

4. For the class of tree sources with context set $\mathcal{T}$ (where, denoting the length of the longest context in $\mathcal{T}$ by m, $|\mathcal{T}| \leq |A|^m$ and typically $|\mathcal{T}| \ll |A|^m$)

$$R_n^* = \frac{|\mathcal{T}|\,(|A|-1)}{2n}\log n + O(n^{-1}). \tag{2.41}$$

2.6 Arithmetic Coding

One way to visualize a discrete distribution over a finite set A is splitting the interval $[0,1)$ into $|A|$ disjoint subintervals open from the right and closed from the left, each corresponding to an element in A so that their lengths are equal to the corresponding probabilities.

If A is a set of symbols, let the intervals be ordered according to the alphabetic order of the symbol set. If A is a set of equally long sequences of symbols, let us order the intervals according to the lexicographic order. A prefix of a sequence always corresponds to a super-interval of the interval corresponding to the original sequence. In other words, an interval corresponding to a given sequence contains as subintervals all the sequences having this one as their prefix.

Finite binary codes attaining equality in (2.9) always correspond to distributions where the probabilities of individual instances are powers of 1/2. Every sequence of

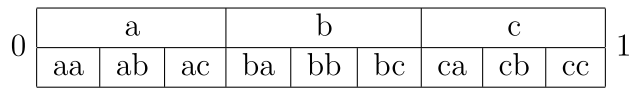

Figure 2.1: I.I.D. source with $P(\mathrm{a}) = P(\mathrm{b}) = P(\mathrm{c}) = 1/3$

binary digits corresponds to an interval that begins at the binary fraction described by this sequence after the binary point, followed by an infinite number of zeroes and ends at the binary fraction described by the same sequence after the binary point followed by an infinite number of ones.

For every infinite sequence of source symbols there is at least one infinite sequence of binary symbols corresponding to a real number $0 \leq r < 1$ described as a binary fraction represented by that sequence after the binary point that lies within the interval corresponding to the infinite source sequence. The converse is not true; this is not a one-to-one correspondence. We can uniquely code infinite source sequences by the infinite binary sequence describing the midpoint (the sum of the infimum and the supremum divided by two) of the set corresponding to the source sequence.

Such an infinite binary code can be decoded in an on-line fashion. For every prefix of the source sequence, there is a finite binary sequence corresponding to an interval that lies fully within the interval corresponding to the source prefix. The reason is that the length of the interval corresponding to the binary code converges to zero as the length of the sequence approaches infinity, while the interval corresponding to any source prefix with non-zero probability has a finite length strictly greater than zero. As we read the code sequence, we can output the source prefix for which the minimal code length has been attained.

Unfortunately, this does not provide us with a one-to-one mapping between all source prefixes and some finite codes, since reading one code bit can result in outputting more than one source symbol. Thus, the length of the prefix we wish to code can be separately encoded using a prefix code on the natural numbers such as Elias Gamma or Fibonacci codes [15]. If the length of the prefix is defined using one of these, the (asymptotic) redundancy rate of the code equals zero. Note that what follows the encoding of the length is a Shannon-Fano code on the n-long source sequences. In practice, we might use a slightly different source model, where the source is a distribution over all possible finite sequences of source symbols. In this variation, the length of the sequence is part of the probabilistic model. Here, we order the source sequences lexicographically and assign intervals to them according to their probability. This is equivalent to the previous case, if we introduce an end-of-string (sentinel) symbol preceding every other symbol in alphabetic order and assign zero probability to all sequences which do not end in an infinite sequence of sentinels. In this case, we do not need to encode the length of the sequence. Instead, we stop as soon as we decode (or encode) the first sentinel symbol, which would guarantee an average code length between H and $H + 1$, where H denotes the entropy of the distribution of the sequences (basically, it is a Shannon-Fano code over the finite sequences).

While at first arithmetic coding seems like the ultimate solution for lossless compression, it is more often than not impractical. There are several reasons for this: One is that we need to provide probabilities for each successive source symbol both during the encoding and the decoding process. Constructing and evaluating a model to such detail is often not practical. Another problem with arithmetic coding is that a direct implementation can run out of numerical precision or buffer space, thus in practice it is implemented with “flush” steps, when the coder pretends that the source sequence has ended, terminates the encoding and (re-)starts coding from the next source symbol. This results in coding redundancy that might not be justified. In general, efficient arithmetic coding implementations are often closely guarded trade secrets or patented methods with high license fees.

The theoretical importance of arithmetic codes is that they can be used to actually achieve the optimal convergence rates from the previous section (see, e.g., [11]).

2.7 Block-Sorting Compressors

Estimating the model parameters of a source with long memory for arithmetic coding is computationally taxing. In 1994, M. Burrows and D. J. Wheeler [6] proposed an alternative approach where the sequence, that can be modeled by a *tree source* (a variety of high-order Markov source; see Section 2.3.4), undergoes a reversible permutation (resorting) after which the resulting sequence is close, in a relative entropy sense, to piecewise I.I.D. with the same entropy as the original sequence. The divergence rate of the resulting sequence from a piecewise I.I.D. sequence with the length of the pieces proportional to the stationary probabilities of the states of the tree source converges to zero, as shown in [14], where the first rigorous proof of the universality of such an approach has also been presented.

It is interesting to note that most block-sorting compressor implementations are not universal for tree sources. On the other hand, even though there is residual redundancy due to the encoding of the transformed sequence that is not entropy-achieving, the convergence rate of such compressors is much faster than that of the theoretically universal Lempel-Ziv algorithm [43, 44] and for the block size of the actual implementation, block-sorting compressors usually outperform L-Z implementations significantly.

The standard architecture of a block-sorting compression algorithm consists of three main components:

$$\boxed{\text{TRANSFORMATION}} \rightarrow \boxed{\text{MODELING}} \rightarrow \boxed{\text{CODING}}$$

Figure 2.2: Components of a block-sorting compression algorithm

The transformation sorts all the suffixes (or prefixes) of the block so that similar contexts are grouped together, and then outputs the characters preceding (or following, respectively) the sorted contexts. It is important to observe that the lexicographical ordering used in the *Burrows-Wheeler Transformation* is by no means the only option. Other sorting methods, that group similar contexts together are viable alternatives. The experimental results in Chapter 3 are obtained using a slightly modified lexicographic ordering, whereby the end-of-file (sentinel) symbol ranks between '0' and '1', rather than preceding '0'.

Modeling the output of the transformation is crucial in obtaining good compression results. In their original paper [6] D. J. Wheeler and M. Burrows use move-to-front (*MTF*) followed by 0-run length encoding as their model. Since then, various different modeling techniques have been devised, such as *MTF2* (Shtarkhov et al.), which is basically a modification of MTF, *Timestamp* modeling [1] (*TS*, by Albers et al.), *Distance Coding* (*DC* by E. Binder, not published) or *Inversion Frequencies* [3] (*IF* by Arnavut and Magliveras). For binary sources, however, most of these techniques are reduced to runlength encoding.

The coding step assigns a binary code to the model. One can use arithmetic codes, Huffman codes or enumerative codes. For the experiments in Chapter 3 we used a Huffman code based on the probabilities from the model. This is the most popular engineering decision in block-sorting compressors. For the binary case, however, enumerative coding also offers several advantages.

2.8 Burrows-Wheeler Transformation

The BWT constitutes a special permutation of the symbols in one block of the original source and is obtained as follows: the set of all cyclic permutations of the source block is sorted in a lexicographic order, and the last characters of the cyclic permutations are recorded. The original string can be retrieved up to a cyclic permutation. There are two common ways of dealing with this ambiguity: either we record the index of the

Input string: `abracadabra$`

Cyclic permutations in lexicographic order:

```
$abracadabra
a$abracadabr
abra$abracad
abracadabra$
acadabra$abr
adabra$abrac        Burrows-Wheeler transform: ard$rcaaaabb
bra$abracada
bracadabra$a
cadabra$abra
dabra$abraca
ra$abracadab
racadabra$ab
```

Figure 2.3: Example of the Burrows-Wheeler Transformation

cyclic permutation that corresponds to the original sequence, or we use a special end-of-string symbol (the sentinel, denoted henceforth by $), in which case the original sequence will be the one ending with the sentinel. In the lexicographic ordering, the sentinel precedes all other symbols. See Figure 2.3 for an example of the BWT.

If we model the input string by an order-m Markov source whereby the probability distribution of each symbol depends on the m subsequent ones (thus, it is a Markov source read from the right to the left), the BWT yields a sequence of strings whose zero-order entropy weighted by their length is equal to the entropy of the Markov source, as it groups the symbols followed by the same context together. This provides for the good compressibility of the BWT.

Since for most natural sources the probability distributions of symbols conditioned on any given context of m symbols have very low entropies, we may very well end up having long runs of identical symbols in the BWT, which is what most modeling techniques try to exploit.

Observe, that if we use a sentinel symbol, the Burrows-Wheeler transformation basically boils down to suffix sorting, which is the lexicographic ordering of all n suffixes $\mathbf{s}_i = x_i x_{i+1} x_{i+2} \dots x_{n-1}$ of the input string $\mathbf{x} = x_0 x_1 \dots x_{n-1}$ $(= \mathbf{s}_0)$. This is a well-studied operation (see, e.g., [4] and [21]), that can be performed in linear time and linear space, albeit with a very large constant factor for both. In practice, therefore, algorithms with $O(n \log n)$ time cost and linear memory requirement (with a much smaller constant factor) are often preferred. Suffix sorting is the most time-consuming operation in block sorting compression.

For a long time, the only known algorithms for linear-time suffix sorting first constructed a *suffix tree* (see Section 2.10) from the input, then traversed the tree in a depth-first order. Modified ordering rules mentioned in Section 2.9.2 can be implemented by modifying the traversal order of the suffix tree.

An alternative algorithm that sorts suffixes directly in linear time without first constructing a suffix tree has been published in 2003 by Kärkkäinen and Sanders [21]. Their approach, inspired by [16], is to sort two lists of suffixes and merge the two lists recursively. First they construct the *suffix array* (a list of indices sorted by the lexicographic order of the corresponding suffixes) of the suffixes starting at positions $i \bmod 3 \neq 0$, by reduction to suffix array construction of a string of two thirds the length (in a procedure they call *lexicographic naming*) recursively. Then, using the result of the first step, a suffix array of the rest of the suffixes is constructed and finally the two are merged. In this algorithm, lexicographic naming is the computational bottleneck.

2.9 Variations of BWT

2.9.1 Schindler Transformation

As noted before, the only property of BWT that we actually use in block-sorting compression is that it groups the symbols with similar contexts together and is invertible. As noted by M. Schindler [37], if we sort the suffixes only according to the first l symbols, the resulting transform will be also invertible, though not as simply as BWT. However, there is a substantial gain in the compression phase, due to a reduced sorting complexity. If we do know in advance (which is often not the case) that the order-l Markov model is adequate for the input, it is often preferable to do the *Schindler Transformation*. However, since in this thesis we aim at a broader, not narrower set of well-compressible sources, we have to rule Schindler's transformation out.

2.9.2 Modified Ordering Rules

As long as we preserve the property that suffixes with identical prefixes are grouped together, the above outlined compressibility analysis applies. Therefore, we are free to choose any ordering that groups identical prefixes together, of which the lexicographic ordering is only one.

A deviation from the lexicographic ordering, however, can provide certain benefits, in case the transition matrix of the Markov model itself exposes regularities. For example, if there are identical or similar rows, it is beneficial to redefine the order of the individual symbols in the lexicographic ordering by placing the rows' corresponding symbols after one another. This way, the adaptive coding for the output will face fewer and less abrupt changes in the 0-order statistics of the output symbols. For example, for natural languages it has been found advantageous to group vowels and consonants together, resulting in the so-called **"aeioubcdgf"** ordering [7].

Another interesting possibility is to reflect the ordering for each second symbol (as in binary reflected Gray ordering). This change will make the first and last suffixes following neighboring letters meet, relieving the adaptive coder of abrupt changes, as these are more likely to be similar than those on the opposite ends of the ordering. Unfortunately, no efficient algorithm inverting such a modified transformation is known. Most algorithms for suffix sorting can be easily modified to sort in this way, with the notable exception of that by Kärkkäinen and Sanders [21].

Both above mentioned aspects of modified orderings have been studies by B. Chapin and S. R. Tate in [7].

2.10 Suffix Tree

A data structure proposed in 1975 by McCreight [28] (as an improvement over Weiner's [41]) to facilitate exact-match substring searching has since then gained significant popularity in string-processing. The *suffix tree* is, in essence, a search index occupying memory space proportional to the length of the string that can be constructed in linear time in the worst case. Finding a substring takes time proportional to the lengths of the substring.

In this section, we give an overview of its important properties, describe the most recent algorithm for its online construction devised by Ukkonen [39]. A good overview of the other two linear-time algorithms and their profound connection to one another and that by Ukkonen is given by Giegerich and Kurtz in their 1996 paper [17].

In 1997, Farach proposed a completely different approach to linear-time suffix tree construction [16]. Farach's paper settled the question about the complexity of suffix tree construction for large alphabets by showing that for integers in the range $[1 \dots n]$, where n is the length of the string, the suffix tree can still be constructed in time proportional to n. This result implies that the time complexity for suffix tree construction equals the time complexity of sorting the symbols in the string, since on one hand the suffix tree construction sorts the symbols as well, while on the other hand, the suffix tree can be constructed by sorting the symbols first and then applying Farach's algorithm.

In Chapter 5, a generalization of this data structure is presented and analyzed, which is one of our principal contributions.

In addition to searching, suffix trees can be used to quickly answer many questions about the corresponding strings and perform various operations on them. In particular, constructing and traversing the suffix tree of a string was the only known method of performing the Burrows-Wheeler transformation in linear time of $|s|$ before the result in [21].

2.10.1 Preliminaries

For any $\mathbf{x} \in A^*$, denote the length of $\mathbf{x}$ (the number of symbols in the sequence) by $|\mathbf{x}|$. The individual symbols in $\mathbf{x}$ are denoted by $x_0, x_1, \ldots, x_{|\mathbf{x}|-1}$. For $0 \leq i \leq j < |\mathbf{x}|$, let x_i^j denote the sequence $x_i, \ldots, x_j$. Let $\mathbf{xy}$ denote the concatenation of $\mathbf{x}$ and $\mathbf{y}$. Finally, let ϵ denote the empty string.

Definition 2.10.1 (Suffix Trie) *A graph on all distinct substrings of* $\mathbf{s}$*, with arcs* $\overrightarrow{\mathbf{x}|\mathbf{y}}$ *iff* $y_0^{|\mathbf{y}|-2} = \mathbf{x}$*. Formally, for every* $\mathbf{s} \in A^*$ *define the directed graph* $T(\mathbf{s}) = (V(\mathbf{s}), E(\mathbf{s}))$ *such that*

$$V(\mathbf{s}) = \{\mathbf{x} \in A^* : \exists i : \mathbf{s}_i^{i+|\mathbf{x}|-1} = \mathbf{x}\} \text{ and } E(\mathbf{s}) = \{\overrightarrow{\mathbf{x}|\mathbf{y}} : \mathbf{x}, \mathbf{y} \in V(\mathbf{s}) \text{ and } \mathbf{x}y_{|\mathbf{y}|-1} = \mathbf{y}\},$$

where $V(\mathbf{s})$ *and* $E(\mathbf{s})$ *are, respectively, the vertex set and the arc set of* $T(\mathbf{s})$.

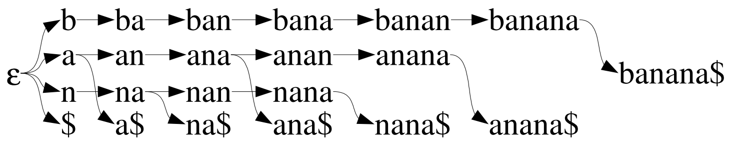

Figure 2.4: Graph T(banana$)

This graph is called the *suffix trie* of $\mathbf{s}$. The following simple statements about $T(\mathbf{s})$ provide the justification for this name. As a data structure, the trie has been first proposed by de la Brandais [5] in 1959. Its name is a word-play on “re*trie*val” and “tree” of unknown origin.

Fact 2.10.1 *Each vertex in* $T(\mathbf{s})$ *has exactly one incoming arc, with the sole exception of* ϵ*, which has none.*

Proof: The only incoming arc to vertex $\mathbf{x}$ points from $x_0^{|\mathbf{x}|-2}$, according to definition 2.10.1. $\square$

Fact 2.10.2 $T(\mathbf{s})$ *is connected.*

Proof: Every vertex $\mathbf{x} \neq \epsilon$ is connected to ϵ by the following path: $\epsilon \to x_0 \to x_0^1 \to \cdots \to \mathbf{x}$. Note that this path is unique due to Fact 2.10.1. $\square$

Fact 2.10.3 $T(\mathbf{s})$ *is cycle-free.*

Proof: $T(\mathbf{s})$ cannot contain a directed cycle, since for each arc $\overrightarrow{\mathbf{x}|\mathbf{y}}$, $|\mathbf{x}| < |\mathbf{y}|$. Thus, the directed cycle cannot continue after the vertex corresponding to the longest sequence. Other kinds of cycles are precluded by Fact 2.10.1. $\square$

Fact 2.10.4 *Every leaf of* $T(\mathbf{s})$ *(except* ϵ*) corresponds to a suffix of* $\mathbf{s}$*. That is* $\forall \mathbf{x} \in V(\mathbf{s}) \setminus \{\epsilon\} : \deg(\mathbf{x}) = 1$ *if* $s_{|\mathbf{s}|-|\mathbf{x}|}^{|\mathbf{s}|-1} = \mathbf{x}$*. If the last symbol of* $\mathbf{s}$ *is unique in* $\mathbf{s}$ *(that is* $\forall i < |\mathbf{s}| - 1 : s_i \neq s_{|\mathbf{s}|-1}$*), the converse statement holds as well; the correspondence is one-to-one.*

Proof: If $\deg(\mathbf{x}) = 1$ and $\mathbf{x} \neq \epsilon$ then the only arc connecting to $\mathbf{x}$ must be the one pointing to it, because of Fact 2.10.1. Therefore, it is a suffix. If the end-of-sequence symbol is unique (denote it by \$), any vertex corresponding to a sequence ending with \$ has no outgoing arcs and one incoming arc. Therefore, it is a leaf. $\square$

Thus, $T(\mathbf{s})$ is a tree (connected, cycle-free graph), as shown by Facts 2.10.2 and 2.10.3, and its leaves correspond to suffixes (Fact 2.10.4).

If we label each arc $\overrightarrow{\mathbf{x}|\mathbf{y}}$ by $y_{|\mathbf{x}|}$, the concatenation of labels along the path from ϵ to any vertex $\mathbf{x}$ provides the sequence corresponding to $\mathbf{x}$. In this representation, the suffix trie can be considered as a finite state deterministic automaton accepting the suffixes of $\mathbf{s}$ and nothing else. The initial state of this automaton is ϵ, the terminal states are those corresponding to suffixes (see Figure 2.5).

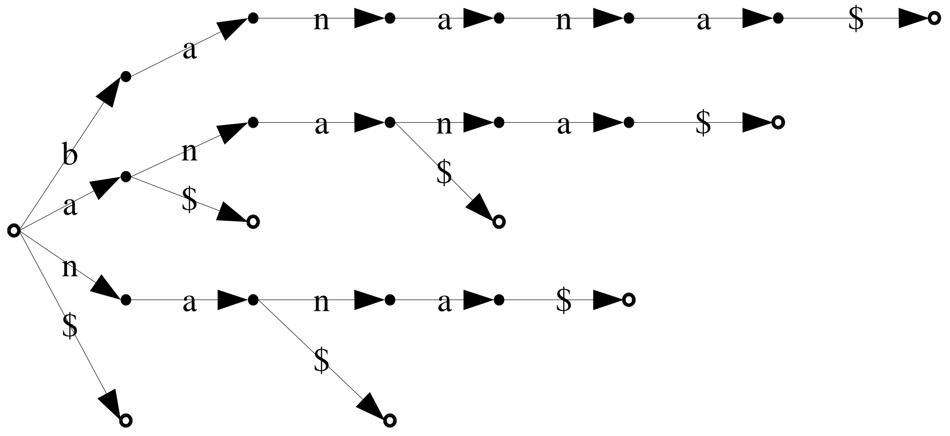

Figure 2.5: Suffix trie of “banana$” as an automaton

2.10.2 Compact Representation

In order to use the suffix trie as a search index, one does not need to store the sequences corresponding to the vertices of $T(\mathbf{s})$, as they are available as the concatenations of arc labels along the (unique) path from ϵ. However, the space requirement for the direct representation of the suffix trie can be super-linear in the length of the sequence. If all the symbols in $\mathbf{s}$ are different, it is easy to see that one needs $\frac{|\mathbf{s}|(|\mathbf{s}|-1)}{2}$ arcs, as there is a $|\mathbf{s}|$-way branching right at the root of the trie, followed by every suffix of $\mathbf{s}$ as a separate branch.

For small alphabets, the situation is not much better: if $|A| \geq 2$ then for any

$c \in \mathbb{R}^+$, there is an $\mathbf{s} \in A^*$ such that

$$|E(\mathbf{s})| > c|\mathbf{s}| \tag{2.42}$$

where $|E(\mathbf{s})|$ denotes the cardinality of the arc set of $T(\mathbf{s})$.

As observed by Weiner [41], by contracting every arc in $T(\mathbf{s})$ which is the only outgoing arc of some vertex, one gets a structure that is linear in $|\mathbf{s}|$. This is the *suffix tree* of $\mathbf{s}$, denoted by $S(\mathbf{s})$. We label the arcs of $S(\mathbf{s})$ by concatenating the labels of the corresponding arcs from $T(\mathbf{s})$ in their order of precedence.

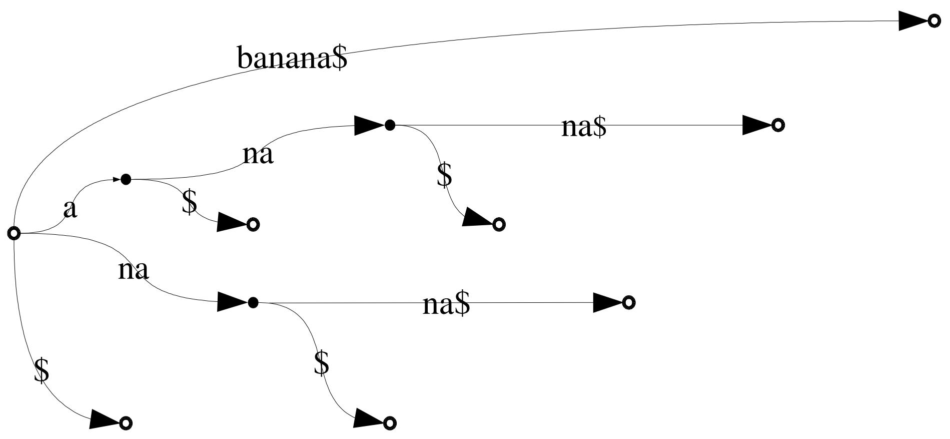

Figure 2.6: Suffix tree of “banana$”

Definition 2.10.2 (Suffix Tree) *The suffix tree of* $\mathbf{s}$ *is a directed graph* $S(\mathbf{s}) = (W(\mathbf{s}), F(\mathbf{s}))$ *where the node set* $W(\mathbf{s}) \subseteq V(\mathbf{s})$ *is defined by*

$$W(\mathbf{s}) = \\ \left\{\mathbf{x} \in A^* : \left(\exists \mathbf{y}, \mathbf{z} \in V(\mathbf{s}) : \overrightarrow{\mathbf{x}|\mathbf{y}}, \overrightarrow{\mathbf{x}|\mathbf{z}} \in E(\mathbf{s})\right) \text{ or } \left(\nexists \mathbf{y} \in V : \overrightarrow{\mathbf{x}|\mathbf{y}} \in E(\mathbf{s})\right) \text{ or } (\mathbf{x} = \epsilon)\right\} \tag{2.43}$$

and the arc set $F(\mathbf{s})$ *is defined as follows:*

For any $\mathbf{x}, \mathbf{y} \in W(\mathbf{s})$, $\overrightarrow{\mathbf{x}|\mathbf{y}}$ *is in* $F(\mathbf{s})$ *iff* $\mathbf{y}_0^{|\mathbf{x}|-1} = \mathbf{x}$ *and there is no index* i *such that* $|\mathbf{x}| \leq i < |\mathbf{y}| - 1$ *and* $\mathbf{y}_0^i \in W(\mathbf{s})$.

Since $S(\mathbf{s})$ is a minor of $T(\mathbf{s})$, it is also a tree. Following Ukkonen [39], we shall call the elements of $W(\mathbf{s})$ *explicit nodes* and those of $V(\mathbf{s}) \setminus W(\mathbf{s})$ *implicit nodes* of the suffix tree.

Assertion 2.10.1 *If* $|\mathbf{s}| > 2$ *then* $S(\mathbf{s})$ *can be represented by an array of less than* $(2 + |A|)|\mathbf{s}|$ *integers.*

2.10.3 Efficient On-line Construction

As mentioned before, there are several known ways of constructing suffix trees in linear time of $|\mathbf{s}|$ [17, 41, 28, 39]. In this section, we will outline Ukkonen's algorithm [39], which has the additional benefit of being an on-line construction. That is, as the input is being read, the suffix tree for the string read so far is always immediately available.

For the purpose of block-sorting, on-line processing is irrelevant, but we nevertheless chose to implement and present Ukkonen's algorithm, as it is, in my opinion, the most intuitive of the three and, as discussed in [17] and mentioned in the original paper by Ukkonen [39] as well, they are all very closely related on the level of implementation.

First, we introduce an auxiliary directed graph over the vertex set V. The set M of arcs (called *suffix link*s following [28]) in this graph is defined as follows:

$$\overrightarrow{\mathbf{x}|\mathbf{y}} \in M \text{ iff there exists } a \in A \text{ such that } a\mathbf{y} = \mathbf{x} \tag{2.44}$$

Observe that the suffix links, if pointed to the opposite direction, constitute a suffix trie of the reversed string, or the so-called *prefix trie.*

Another important observation is that *suffix links from explicit nodes point to explicit nodes.* By pointing these arcs in the opposite direction, we can obtain the *prefix tree*; the suffix tree of the reversed string [28].

If the prefix tree is viewed as the objective, Ukkonen's algorithm is essentially identical to the prefix tree construction algorithm derived from McCreight's algorithm by M. Effros, as presented in [13]. This just underlines the deep-running similarity between these two seemingly different algorithms, which is also mentioned in the introduction of Ukkonen's original paper [39].

Definition 2.10.3 (Boundary Path) *The path of suffix links from* $\mathbf{s}$ *to* ϵ *is called the* boundary path *of* $T(\mathbf{s})$.

For technical purposes, the following features are also added to the suffix tree: leaf nodes have a special end pointer to the end of the string (which can be "infinity" if the length is not known in advance) and there is an additional explicit node added before the *root* of the tree (the node corresponding to ϵ) denoted (following [39]) by $\perp$. During initialization, a suffix link $\overrightarrow{\epsilon|\perp}$ is added to M and multiple arcs $\overrightarrow{\perp|\epsilon}$ labeled with all elements of A are added to $F(\mathbf{s})$.

The on-line construction of the suffix tree is derived from a simple and intuitive algorithm constructing the suffix trie $T(\mathbf{s})$ in an on-line fashion as follows. In order to construct $T(s_0^k)$ from $T(s_0^{k-1})$, the boundary path of $T(s_0^{k-1})$ is traversed and a new node pointed by an arc labeled with s_k is added, if there isn't one already. The nodes to which s_k-labeled arcs point from the boundary path of $T(s_0^{k-1})$ are connected with suffix links in the order of traversal. These suffix links will form the boundary path of $T(s_0^k)$. This iteration is repeated until $T(\mathbf{s})$ is ready.

Note that if there is an outgoing arc in $T(\mathbf{s})$ labeled with some $a \in A$ from a node on the boundary path, then the node pointed by an outgoing suffix link, if it exists, also must have an outgoing arc labeled with a. This, in turn, implies that the traversal of the boundary path in the above iteration can be stopped when the first outgoing arc labeled with s_k is found on the boundary path of $T(s_0^{k-1})$, leading to the following definition:

Definition 2.10.4 (Active Point) *The first node on the boundary path of* $T(s_0^{k-1})$ *is called* active point *if it has an outgoing arc labeled with* s_k.

Another important observation is that leaf nodes (all of which obviously lie on the boundary path) will have a new link and a new node added in each step, which will become a new leaf node. Thus, when adding a leaf, one can add the whole path to the end of **s**, avoiding the extension of each leaf in every iteration. The first branching along the boundary path can only occur at one of the successors of the active point.

The simplifications made possible by the above two observations and the addition of $\perp$ to each boundary path (making sure that there is always an active point) result in each iteration starting at the active point of $T(s_0^{k-1})$ and ending at the active point of $T(s_0^k)$. Using the compact representation of Section 2.10.2 and this simplified algorithm, we obtain a linear, on-line algorithm for the construction of $S(\mathbf{s})$.

A more formal, step-by-step description of the generalized version of this algorithm is presented in Chapter 5.

Chapter 3

A Simple Binary Block-Sorting Compressor and its Experimental Evaluation

In this chapter, the experimental evaluation of a block-sorting compression scheme is presented that operates on the bit level. It shows that even such a simple technique, which ignores byte boundaries and uses a very simple modeling scheme for the output of the block-sorting transform, outperforms some of the best industry standard compressors for sources that are not byte-aligned, while providing reasonable compression ratios for byte-aligned sources. These preliminary results can be substantially improved using more sophisticated modeling and coding techniques. However, the obtained experimental results point out the potential advantages of bit-level compression.

The results presented in this chapter have been published in the proceedings of DCC 2003 [30].

3.1 Binary Block-Sorting

In binary block sorting, the fact that we have to deal with only three symbols, one of which (the end-of-string or sentinel, symbol \$) occurs only once in the file offers many opportunities for simplifying the techniques of general block-sorting. Essentially, all common suffix sorting techniques become a lot simpler if applied to binary sequences, particularly the only known linear one at the time of performing this evaluation, which builds a suffix tree and then traverses it. Since then, Kärkkäinen and Sanders published a direct suffix array construction [21] that also takes linear time.

Before elaborating on the one we used, let us point out how other sorting techniques may benefit from a binary alphabet:

- *Bucket sort* steps can be done in place by simply moving all suffixes that begin with 0 to the beginning and then exchanging the next element with the empty suffix (containing \$ only).

- The dependence of many simple algorithms' running time on the number k of symbols is linear and slightly superlinear on the file length n, typically $O(kn \log n)$. Thus, when working with bits instead of bytes, even the *8-fold increase* in file length does not overweight the *256-fold decrease* in symbol set cardinality.

- The most popular *three-way quicksort* [4] (by J. L. Bentley and R. Sedgewick) reduces to binary radix sort, yielding a far simpler (and faster) code.

The only drawback of building a suffix tree first is the significantly increased memory-overhead. The direct algorithm in [21] would have been better in this respect. It is important to emphasize that this memory overhead is strictly linear with file length.

For experimental purposes we have implemented Ukkonen's *online suffix tree construction* algorithm [39], which does not deal with the sentinel until the very last step, thus keeping the suffix tree in binary form. Moreover, it is a full binary tree in the sense that a node is either a leaf or a two-way branching. Therefore, it can be efficiently stored, without hashing or other workaround solutions that k-ary suffix tree implementations often suffer from, rendering them unfeasible for practical BWT. Each inner node can be represented by two pointers into the string marking the beginning and the end of the string segment associated with the node, two child pointers and one suffix link (needed by Ukkonen's algorithm). This makes an inner node 5 pointers long, which in today's 32-bit computers translates into 160 bits. Leaf nodes can be represented by an even smaller data structure, but we haven't done that optimization. This way, in worst case we have a 320-fold overhead, which is still very large, but considering the available computing power it is well within the realm of feasible solutions.

The binary tree structure can be retained even when reaching the sentinel. Observe that when the sentinel is added to the suffix tree, it is either on the already existing leaves or it causes a new branching and forms a leaf of its own. Observe, furthermore, that a node's end-of-string pointer can point to the last character of the input string if and only if it has the sentinel as its child. Thus, we know exactly which inner nodes have the sentinel as a child, so there's no need to actually represent it. At this last step we may actually violate the "full" tree invariant, as some of the sentinel leafs may not have both 0 and 1 siblings. A more detailed description and analysis of this implicit end-of-string representation is provided in Section 5.3.

Example:

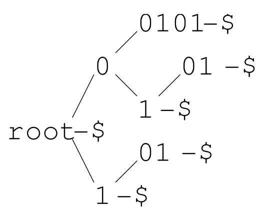

Figure 3.1: Suffix tree of `00101$`

After building the tree, we can traverse it in a depth-first manner outputting the appropriate character of the BWT each time we encounter a sentinel leaf. The sentinel itself is outputted (or more precisely, noted), when we traverse the root node. For a Gray ordering (see Section 2.9.2), one can change the traversal direction each time we encounter a 1. For the experiments presented in this chapter, we have not implemented this.

3.2 Modeling and Encoding

The modeling of the transformation's output is a complicated issue on which the efficiency of the block-sorting compressor ultimately depends. However, even simple models provide for surprisingly good results, which largely explains the popularity of BTW with the data compression community.

For experimental purposes we have implemented an extremely simple model that approximates the 0-order entropy of the counts of successive identical symbols (a.k.a. *runlength*), which obviously does not attain the *entropy of the binary Markov source*, which in turn falls short of the *entropy of the Markov source based on multibit symbols.* Thus, our model is blatantly suboptimal, so it is not surprising that it does not compete with a finely tuned block-sorting compressor like `bzip2` when the latter's model assumptions are met. However, its performance is still satisfactory in these cases, and comfortably surpasses that of `bzip2`, when the symbol length is not 8 bits.

Runs of b identical bits occur, when there are b identical "words" (bit sequences of arbitrary length). As B. B. Mandelbrot has observed in [27], the number of "word" occurrences is drawn from a *hyperbolic distribution* (see also *Zipf's law* [20]), under certain, very general assumptions on the "text". However, since the first bits of two different words can be the same (which can be modelled by a Bernoulli random variable), the actual length of a run is $L = \sum_{i=1}^{G} Z_i$, where Z_i's are random variables drawn from a hyperbolic distribution, while G is drawn form a geometric distribution. This yields a "displaced" hyperbolic distribution:

$$P(L = x) = c\left(\frac{1+a}{x+a}\right)^b$$

whereby $c > 0$ is just a normalization constant, $a \geq 0$ is the displacement parameter, and $b \geq 1$ is the parameter of the hyperbolic distribution; $a, b, c \in \mathbb{R}$.

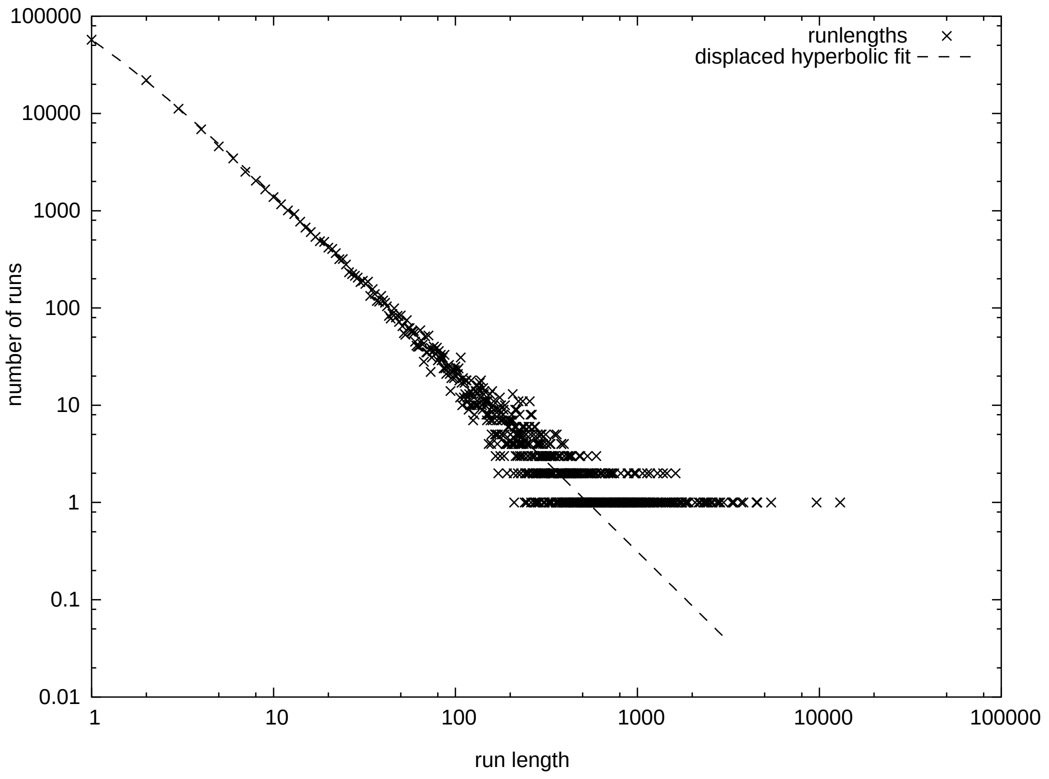

Figure 3.2: Runlength distribution of the binary BWT of an essay in English

We estimate this distribution using a robust linear fit on the logarithmic scale. Using this estimate, a Huffman code (see, e.g., [8, Section 5.6]) is constructed to encode the runlengths.

3.3 Experimental Results

Even though there are many types of not byte-aligned data, such as *Morse-coded messages*, *genetic sequences* (where each base pair is represented by 2 bits), *7-bit ascii texts* without the 8th padding bit and many more, on which the presented compressor outperforms those that assume byte aligned input, there is no standard corpus of such sources. Therefore, we compressed each file in *Calgary corpus*[42] with 0-order Huffman code (including the Huffman table for decodability), and used these as examples of files with variable symbol length.

Table 3.1 summarizes the results of our experiments. On the left, we present the compression results on the original Calgary corpus, while on the right those on the Huffman-compressed files (hence the .h8 extension). Observe, that our compressor compressed the original and the Huffman coded files to just about the same length, reflecting that their entropies (that have not been attained, due to the suboptimal modeling) are roughly identical. The two industry standard compressors, by contrast, compressed the not byte-aligned versions to longer files, reflecting that they used a non-fitting source model. The lengths of the compressed output of `gzip` are provided only as a baseline, as it is not a block-sorting compressor.

NAME	length	bzip2	gzip	ours	NAME	length	bzip2	gzip	ours
bib	111261	27467	34896	32022	bib.h8	72836	48985	52223	33418
book1	768771	232598	312275	242857	book1.h8	438449	327635	396817	250069
book2	610856	157443	206152	170783	book2.h8	368375	247470	294393	176314
geo	102400	56921	68410	66370	geo.h8	72718	69747	71318	64585
news	377109	118600	144395	135444	news.h8	246471	186895	202768	141263
obj1	21504	10787	10315	12727	obj1.h8	16190	14457	13551	12612
obj2	246814	76441	81082	98395	obj2.h8	194227	137073	139887	99407
paper1	53161	16558	18536	19816	paper1.h8	33411	28079	27720	20241
paper2	82199	25041	29660	28084	paper2.h8	47689	39592	40630	29158
paper3	46526	15837	18067	18124	paper3.h8	27346	24439	24314	18646
paper4	13286	5188	5527	6047	paper4.h8	7931	7830	7328	6124
paper5	11954	4837	4988	5815	paper5.h8	7507	7447	6937	5762
paper6	38105	12292	13206	14786	paper6.h8	24097	20888	20246	15206
pic	513216	49759	52377	59131	pic.h8	106712	58819	58469	52729
progc	39611	12544	13255	15320	progc.h8	25989	21877	21090	15815
progl	71646	15579	16158	18101	progl.h8	43056	30219	29279	19274
progp	49379	10710	11180	13336	progp.h8	30288	22236	21429	13716
trans	93695	17899	18856	22864	trans.h8	65302	40714	40004	23563

Table 3.1: Compression results on the Calgary Corpus

3.4 Conclusions

As we have expected, the bit-based compressor significantly outperformed the byte based ones on data which was not aligned to byte boundaries. It is also worth noting that it managed to outperform the LZ-77-based `gzip` on some byte aligned inputs, which is due to the faster convergence of the block-sorting compression technique.

When compared to the byte-based block-sorting bzip2, the bit-based compressor performed worse on byte-aligned data. Part of it is because of the inherent redundancy resulting from an inaccurate source model (tree source on bits rather than a tree source on bytes), part of it is a result of crude modeling of the transformed data. But which of the two is dominant? Can we do much better by improving the post-processing of the BWT? This is a very important question into which the next chapter will give some insight from a theoretical point of view.

Another interesting question is to explain in a quantitative manner the poor performance of byte-based models on data that is not aligned to byte boundaries.

It is worth mentioning that the experimental compressor described in this chapter routinely compresses JPEG images further by approximately 10%, which is quite surprising in the light of the fact that JPEG files have already undergone lossless compression.

Chapter 4

Information Theoretical Results

In this chapter, information theoretic findings are presented that help in explaining what has been observed in the experiments described in the previous chapter. In particular, we shall prove an upper bound on redundancy one has to pay for not taking the block nature of the source into account in the statistical model. Some of the results in this chapter are joint work with András György and have been presented at two conferences [31, 32]. A joint paper [33] with these results has been submitted for publication in IEEE Transactions on Information Theory.

4.1 Preliminaries

A binary block code of length n for the source alphabet A is given by a function $f_n : A^n \to \{0,1\}^*$, which maps any source vector $x \in A^n$ to the binary string $f(x)$. The length function $\ell_n : A^n \to \mathbb{N}$ associated with f_n gives for each x the length of the corresponding binary string, that is, $\ell_n(x) = |f_n(x)|$. We require f_n to be uniquely decodable, that is, for $x_1, \ldots, x_j, y_1, \ldots, y_k \in A^n$, $f_n(x_1)f_n(x_2)\ldots f_n(x_j) = f_n(y_1)f_n(y_2)\ldots f_n(y_k)$ if and only if $j = k$ and $x_i = y_i, i = 1, \ldots, j$, where for two binary strings s_1 and s_2, s_1s_2 denotes their concatenation. It is well known [8] that if f_n is uniquely decodable than its length function ℓ satisfies the Kraft inequality

$$\sum_{x \in A^n} 2^{-\ell_n(x)} \leq 1.$$

Moreover, for any such code there exists a prefix code with the same length function, and also there exists another prefix code f'_n with length function ℓ'_n such that $\ell'_n(x) \leq \ell_n(x)$ for all $x \in A^n$, and the equality holds for ℓ'_n in the Kraft inequality, that is, $\sum_{x \in A^n} 2^{-\ell'_n(x)} = 1$. Therefore, without loss of generality, in the rest of the chapter we consider only codes for which the Kraft inequality holds with equality. Therefore, the *coding distribution* of f_n, defined as

$$P_{f_n}(x) = 2^{-\ell_n(x)}$$

for each $x \in A^n$, is a proper probability distribution.

The redundancy of the code f_n with length function ℓ_n for the random vector X_0^{n-1} is defined as

$$R_n = E\ell_n(X_0^{n-1}) - H(X_0^{n-1}) = E\left(\ell_n(X_0^{n-1}) + \log P_{X_0^{n-1}}(X_0^{n-1})\right)$$

the difference of the expected code length $E\ell_n(X_1^n)$ and the entropy

$$H(X_0^{n-1}) = -\sum_{x \in A^n} P_{X_0^{n-1}}(x) \log P_{X_0^{n-1}}(x).$$

Note that $R_n \geq 0$, and if Y_0^{n-1} is distributed according to P_{f_n}, then

$$R_n = D(X_0^{n-1} \| Y_0^{n-1}).$$

Similarly, for any distribution P_n over A^n, one can construct a prefix code with length function $\ell'_n(x) = -\lceil \log P_n(x) \rceil$. The redundancy of this code can be bounded as

$$R'_n = E\ell'_n(X_0^{n-1}) - H(X_0^{n-1}) \leq D(X_0^{n-1} \| \widehat{Y}_0^{n-1}) + 1$$

where $\widehat{Y}_0^{n-1}$ is distributed according to P_n.

A binary source code for an infinite source X_0^∞ taking values in the alphabet A is given by a sequence of block-n codes f_n. Without loss of generality we assume that for each f_n equality holds in the Kraft inequality. If the coding distributions P_{f_n} are compatible in the sense that there is an A-valued random process Y_0^∞ such that the distribution of Y_0^{n-1} is P_{f_n} for all n, then the redundancy rate of the code is given as

$$\lim_{n\to\infty} \frac{1}{n} R_n = \lim_{n\to\infty} \frac{1}{n} D(X_0^{n-1} \| Y_0^{n-1}) = \bar{D}(X_0^\infty \| Y_0^\infty)$$

provided the limit exists [8],[11]. If X_0^∞ is a block-N stationary block-N Markov source and Y_0^∞ is a stationary mth order Markov source, then both sources are block stationary block-mN Markov sources; for such sources the limit always exists [19].

In the sequel we will alternately use the *code* for either a block-n code f_n, or a sequence of codes $\{f_n\}_{n=1}^\infty$.

4.2 Approximation of block-Markov sources

Here we consider the following question: Suppose that we want to encode a given block-N Markov source using a code with a coding distribution that is mth order Markov. What is the minimum coding redundancy if the mth order Markov model is optimally chosen?

In view of the previously discussed equivalence between redundancy and divergence, what we really want to find is the best mth order Markovian approximation of a block-N stationary block-N Markov source X_0^∞ in the divergence sense. That is, we look for an mth order Markov source Y_0^∞ achieving the minimum

$$\bar{D}_m \triangleq \min\big\{\bar{D}(X_0^\infty \,\|\, Y_0^\infty) \,:\, Y_0^\infty \text{ is } m\text{th order Markov}\big\}.$$

Clearly, without loss of generality we may assume that Y_0^∞ is stationary.

Let $\{X_n\}_{n=-\infty}^\infty$ be the two-sided block-N stationary extension of $\{X_n\}_{n=0}^\infty$, and let $\{Y_n\}_{n=-\infty}^\infty$ be the two-sided stationary extension of $\{Y_n\}_{n=0}^\infty$. The minimizing $\{Y_n\}$ and the minimum divergence rate will be expressed in terms of the random variables

$$U_j = X_{j-m+\tau}, \quad j = 0, 1, 2, \ldots$$

where τ is a random variable that is uniformly distributed on $\{0, 1, \ldots, N-1\}$ and is independent of $\{X_n\}$. Notice that $\{U_j\}$ can be seen as a stationary version of the (only) block-N stationary source $\{X_n\}$. With this in mind, it is intuitively clear that the best mth order Markovian approximation of $\{U_n\}$, which has the same mth order conditional distributions as $\{U_n\}$, will also be the best approximation for $\{X_n\}$. This statement is formalized in the next theorem.

Theorem 4.2.1 *Given a block-N Markov source X_0^∞, the relative entropy rate $\bar{D}(X_0^\infty \,\|\, Y_0^\infty)$ is minimized over all stationary mth order Markov sources Y_0^∞ if and only if $P_{Y_m|Y_0^{m-1}} = P_{U_m|U_0^{m-1}}$. The minimum relative entropy rate is given for all $m \geq 2N$ by*

$$\bar{D}_m = I(\tau; U_m | U_0^{m-1})$$

the conditional mutual information between τ and U_m given U_0^{m-1}. Moreover, there is a stationary version $\widehat{Y}_0^\infty$ of Y_0^∞ such that $P_{\widehat{Y}_0^m} = P_{U_0^m}$.

Expressing conditional mutual information in terms conditional entropies as

$$I(\tau; U_m | U_0^{m-1}) = H(\tau | U_0^{m-1}) - H(\tau | U_0^m)$$

we obtain

$$\begin{aligned}
&\sum_{m=2N}^{\infty} I(\tau; U_m | U_0^{m-1}) \\
&\quad = \sum_{m=2N}^{\infty} \Big(H(\tau | X_{\tau-m}^{\tau-1}) - H(\tau | X_{\tau-m}^{\tau}) \Big) \\
&\quad \le H(\tau | X_{\tau-2N}^{\tau-1}) - \liminf_{m\to\infty} H(\tau | X_{\tau-m}^{\tau}) \le \log N
\end{aligned}$$

where the first inequality follows since we clearly have $H(\tau | X_{\tau-m-1}^{\tau-1}) = H(\tau | X_{\tau-m}^{\tau})$.

Thus we obtain the following corollary which states that the block-Markov source can be arbitrarily closely approximated by higher-order Markov models by increasing the model order.

Corollary 4.2.1 *The minimum relative entropy rate $\bar{D}_m$ satisfies*

$$\sum_{m=2N}^{\infty} \bar{D}_m \le \log N.$$

In particular

$$\lim_{m\to\infty} \bar{D}_m = 0.$$

Remark The fact that $\bar{D}_m$ converges to zero as $m \to \infty$ is not very surprising in view of the fact that the divergence rate between a stationary process and its best mth order Markov approximation asymptotically vanishes as $m \to \infty$ (see, e.g., [19]). Note, however, that X_0^{∞} is non-stationary, and that the theorem gives an explicit expression for the optimum approximating process and a characterization of the resulting minimum divergence rate $\bar{D}_m$. In the next section we will use this result to determine the rate at which $\bar{D}_m$ converges to zero.

Proof of Theorem 4.2.1 For all $n > m$ we have from the chain rule for the relative entropy [8]

$$\begin{aligned} &D(X_0^n \,\|\, Y_0^n) \\ &\quad = \sum_{i=m}^{n} D(X_i|X_0^{i-1} \,\|\, Y_i|Y_0^{i-1}) + D(X_0^{m-1} \,\|\, Y_0^{m-1}) \end{aligned}$$

where

$$D(X_i|X_0^{i-1}\|Y_i|Y_0^{i-1}) = \sum_{a_0^i \in A^{i+1}} P_{X_0^i}(x_0^i) \log \frac{P_{X_i|X_0^{i-1}}(a_i|a_0^{i-1})}{P_{Y_i|Y_0^{i-1}}(a_i|a_0^{i-1})}.$$

Observe that if $m \geq 2N$, then for any $i \geq m$,

$$P_{X_i|X_0^{i-1}}(\cdot|x_0^{i-1}) = P_{X_i|X_{i-m}^{i-1}}(\cdot|x_{i-m}^{i-1})$$

and

$$P_{Y_i|Y_0^{i-1}}(\cdot|y_0^{i-1}) = P_{Y_m|Y_0^{m-1}}(\cdot|y_0^{m-1}).$$

Therefore

$$\begin{aligned} D\Big(X_i|X_0^{i-1} \,\|\, Y_i|Y_0^{i-1}\Big) &= \sum_{a \in A^i} P_{X_0^{i-1}}(a) D\Big(X_i|X_0^{i-1} = a \| Y_i|Y_0^{i-1} = a\Big) \\ &= \sum_{b \in A^m} P_{X_{i-m}^{i-1}}(b) D\Big(X_i|X_{i-m}^{i-1} = b \| Y_m|Y_0^{m-1} = b\Big) \\ &= \sum_{b \in A^m} P_{X_{t-m}^{t-1}}(b) D\Big(X_t|X_{t-m}^{t-1} = b) \| Y_m|Y_0^{m-1} = b\Big) \end{aligned}$$

where $t = i \mod N$. Denoting the last sum by S_t, we obtain

$$\lim_{n\to\infty} \frac{1}{n+1} D(X_0^n\|Y_0^n) = \lim_{n\to\infty} \frac{1}{n+1} \sum_{i=m}^{n} D(X_i|X_0^{i-1} \,\|\, Y_i|Y_0^{i-1}) = \frac{1}{N} \sum_{t=0}^{N-1} S_t.$$

Let τ denote a uniform random variable over $\{0, 1, \ldots N-1\}$ that is independent of the pair $(\{X_n\}, \{Y_n\})$, and define the random vectors $U_0^m = X_{\tau-m}^{\tau}$ and $V_0^m = Y_{\tau-m}^{\tau}$.

Then we can rewrite the relative entropy rate as

$$\begin{aligned}
&\bar{D}(X_0^\infty \| Y_0^\infty) \\
&\quad = \sum_{t=0}^{N-1} P_\tau(t) \sum_{b\in A^m} P_{U_0^{m-1}|\tau}(b \mid t) \\
&\qquad \cdot D\left(U_m \mid U_0^{m-1} = b, \tau = t \,\|\, V_m \mid V_0^{m-1} = b, \tau = t\right) \\
&\quad = \sum_{t=0}^{N-1} P_\tau(t) \sum_{b\in A^m} P_{U_0^{m-1}|\tau}(b \mid t) \\
&\qquad \cdot \sum_{x\in A} P_{U_m|U_0^{m-1},\tau}(x \mid b, t) \log \frac{P_{U_m|U_0^{m-1},\tau}(x \mid b, t)}{P_{Y_m|Y_0^{m-1}}(x \mid b)} \\
&\quad = \sum_{t=0}^{N-1} \sum_{b\in A^m} \sum_{x\in A} P_{U_0^m,\tau}(b, x, t) \\
&\qquad \cdot \log \frac{P_{\tau|U_0^m}(t \mid b, x)\, P_{U_m|U_0^{m-1}}(x \mid b)}{P_{Y_m|Y_0^{m-1}}(x \mid b)\, P_{\tau|U_0^{m-1}}(t \mid b)} \\
&\quad = \sum_{t=0}^{N-1} \sum_{b\in A^m} \sum_{x\in A} P_{U_0^m,\tau}(b, x, t) \log \frac{P_{\tau|U_0^m}(t \mid b, x)}{P_{\tau|U_0^{m-1}}(t \mid b)} \\
&\qquad + \sum_{t=0}^{N-1} \sum_{b\in A^m} \sum_{x\in A} P_{U_0^m,\tau}(b, x, t) \log \frac{P_{U_m|U_0^{m-1}}(x \mid b)}{P_{Y_m|Y_0^{m-1}}(x \mid b)}.
\end{aligned}$$

Observe that only the second term of the last expression depends on the choice of $\{Y_n\}$. Since this term is equal to $D(U_m|U_0^{m-1} \| Y_m|Y_0^{m-1})$ (so it is nonnegative), it is uniquely minimized by the choice $P_{Y_m|Y_0^{m-1}} = P_{U_m|U_0^{m-1}}$. With this optimum choice the second term vanishes, so

$$\begin{aligned}
\bar{D}_m &= \sum_{t=0}^{N-1} \sum_{b\in A^m} \sum_{x\in A} P_{U_0^m,\tau}(b, x, t) \log \frac{P_{\tau|U_0^m}(t \mid b, x)}{P_{\tau|U_0^{m-1}}(t \mid b)} \\
&= \sum_{t=0}^{N-1} \sum_{b\in A^m} \sum_{x\in A} P_{U_0^m,\tau}(b, x, t) \log P_{\tau|U_0^m}(t \mid b, x) \\
&\quad - \sum_{t=0}^{N-1} \sum_{b\in A^m} P_{U_0^{m-1},\tau}(b, t) \log P_{\tau|U_0^{m-1}}(t \mid b) \\
&= H\left(\tau \mid U_0^{m-1}\right) - H\left(\tau \mid U_0^m\right) = I\left(\tau; U_m | U_0^{m-1}\right)
\end{aligned}$$

which was to be shown.

Finally, as $P_{Y_m|Y_0^{m-1}} = P_{U_m|U_0^{m-1}}$ and U_0^∞ is stationary, starting the mth order Markov chain Y_0^∞ from the distribution $P_{U_0^{m-1}}$ results in a stationary version of Y_0^∞. This proves the last statement of the theorem. $\square$

From a coding point of view, Theorem 4.2.1 states that if a coding procedure is optimal for Y_0^∞ (in the sense that its length functions correspond to the marginal distributions of Y_0^∞), then it can compress X_0^∞ with rate not exceeding the source entropy rate $\bar{H}(X_0^\infty) = \lim_{n\to\infty} H(X_0^{n-1})$ by more than $\bar{D}_m$. However, in practical situations such codes are not available, as the distribution of Y_0^∞ is usually not known. Moreover, as the triangle inequality does not hold for divergences[1], a code which is almost optimal for Y_0^∞ need not be good at all for X_0^∞. Still, it is reasonable to expect that codes that are universal for the class of mth order Markov sources (that is, perform asymptotically optimally for all sources in the class, including Y_0^∞) will perform well on X_0^∞. This will be shown (together with convergence rates) in Section 4.4.

4.3 Rate of convergence

In this section we examine the rate of convergence at which the minimum relative entropy rate $\bar{D}_m$ converges to 0 in Corollary 4.2.1. In fact, we will show that $\bar{D}_m$ vanishes exponentially fast, that is, a block-Markov source can be very well approximated by high order Markov sources.

From Theorem 4.2.1 we can see that in order to establish that rate of convergence, it is sufficient to estimate the conditional entropy $H(\tau|U_0^m)$. Using Fano's inequality (see, e.g., [8]) we will trace back our problem to the problem of classification of Markov sources. In this latter problem, given finitely many Markov sources, one has to decide

[1]This prevents us from establishing the upper bound on the redundancy rate of a code applied to X_0^∞ as the sum of the divergence rate between X_0^∞ and Y_0^∞ and the divergence rate between Y_0^∞ and the coding distribution.

which one of them has generated an observed sequence. In previous works it was shown that, under various conditions, this problem can be solved with exponentially decaying error probability as the length of the observed sequence increases, see, e.g., [34, 2, 29]. However, the conditions in these works are not immediately applicable to our setup. Therefore, first we revisit some results from Csiszár *et al.* [10] concerning large deviations of Markov chains. Based on these results, in Lemma 4.3.2 we derive an upper bound on the classification error (using a similar method as in [34]).

Let A be a finite set, and let Λ denote the set of distributions over A^2. The second order type of a sequence $u_0^n = (u_0, \ldots, u_n) \in A^{n+1}$ is the empirical distribution of the pairs (u_k, u_{k+1}) over A^2 defined by the relative frequencies

$$P^{(2)}_{u_0^n}(v_1, v_2) = \frac{1}{n}|\{k \in \{0, \ldots, n-1\} : u_k = v_1, u_{k+1} = v_2\}|, v_1, v_2 \in A.$$

For any distribution $P(u, v)$ over A^2, let $\bar{P}(u) = \sum_{v \in A} P(u, v)$ denote the marginal distribution of the first coordinate, and for $\bar{P}(u) > 0$, let $P(v|u) = P(u, v)/\bar{P}(u)$. For any stochastic matrix $\{W(v|u)\}_{u,v \in A}$ (that is, $W(v|u) \geq 0$ for all $u, v \in A$, and $\sum_{v \in A} W(v|u) = 1$), let

$$D(P\|W) = \sum_{u,v} P(u, v) \log \frac{P(u, v)}{\bar{P}(u)W(v|u)} = \sum_{u,v} P(u, v) \log \frac{P(v|u)}{W(v|u)}$$

denote the relative entropy between the distributions $P(u, v)$ and $\bar{P}(u)W(v|u)$. For any set of distributions $\Pi \subset \Lambda$ let $cl\Pi$ denote its closure in Λ under point-wise convergence. Finally, let $\Lambda_0 = \{P \in \Lambda : \sum_{v \in A} P(u, v) = \sum_{v \in A} P(v, u), u \in A\}$ denote the distributions in Λ with equal marginals, and for any stochastic matrix W, let $S(W) = \{P \in \Lambda : P(u, v) = 0 \text{ if } W(v|u) = 0, u, v \in A\}$. The following lemma is proved in [10].

Lemma 4.3.1 ([10, Lemma 2a]) *Assume that $\{X_i\}$ is a Markov chain with finite* *alphabet A and transition matrix W, and let $\Pi \subset S(W)$ be arbitrary. Then for every $u \in A$,*

$$\limsup_{n\to\infty} \frac{1}{n} \log \Pr(P^{(2)}_{X_0^n} \in \Pi | X_0 = u) \le - \min_{P \in \Lambda_0 \cap cl\Pi} D(P\|W).$$

Note that the minimum on the right hand side of the above inequality is attainable, as it is the minimum of a continuous function over a compact set.

Following the approach of Natarajan [34], this result easily leads to classification of Markov sources. Assume that the sample $X_1, X_2, \ldots, X_n$ is generated with equal probability by one of K stationary Markov sources with transition matrices $W_1, \ldots, W_K$, respectively. The problem is to determine which source has generated the sample. The next lemma provides a classification method for irreducible Markov-chains with exponentially decaying error probability as the sample size grows. (A Markov-chain with transition matrix W is called irreducible if for every pair $(u, v) \in A^2$ there is a positive integer n such that the element in the (u, v) position of W^n is positive.)

Lemma 4.3.2 *Let $\{X_{i,n}\}_{n=0}^{\infty}$, $i = 1, \ldots, K$, $K \ge 2$, be independent Markov sources with irreducible transition matrices W_i such that $W_i \ne W_j$ for $i \ne j$. Assume that t is distributed over $\{1, \ldots, K\}$ such that $\Pr(t = i) > 0$ for all $i = 1, \ldots, K$, and t is independent of the $\{X_{i,n}\}$'s. Finally, assume that we observe the tth Markov source, that is, let $X_n = X_{t,n}$ for $n = 0, 1, \ldots$. Define*

$$R_i = \{P \in \Lambda : D(P\|W_i) < D(P\|W_j) \text{ for all } j \ne i\},$$

and let $\hat{t}_n = i$ if $P^{(2)}_{X_0^n} \in R_i$ for some $i \in \{1, \ldots, K\}$ and let $\hat{t}_n$ be arbitrary otherwise. Then for any $u \in A$,

$$\limsup_{n\to\infty} \frac{1}{n} \log \Pr(t \ne \hat{t}_n | X_0 = u) \le - \min_{1 \le i \le K} \min_{P \in \Lambda_0 \cap \bar{R}_i} D(P\|W_i) < 0$$

where $\bar{R}_i = \Lambda \setminus R_i$ denotes the complement of R_i.

Proof. It is easy to see that

$$\begin{aligned}
\Pr(t \neq \hat{t}_n | X_0 = u) &= \sum_{i=1}^{K} \Pr(t = i | X_0 = u) \Pr(\hat{t}_n \neq i | t = i, X_0 = u) \\
&= \sum_{i=1}^{K} \Pr(t = i | X_0 = u) \Pr(\hat{t}_n \neq i | t = i, X_{i,0} = u) \\
&\leq \sum_{i=1}^{K} \Pr(t = i | X_0 = u) \Pr(P^{(2)}_{X^n_{i,0}} \in \bar{R}_i | X_{i,0} = u) \\
&\leq \max_i \Pr(P^{(2)}_{X^n_{i,0}} \in \bar{R}_i | X_{i,0} = u) \\
&= \max_i \Pr(P^{(2)}_{X^n_{i,0}} \in \bar{R}_i \cap S(W_i) | X_{i,0} = u).
\end{aligned}$$

Here the last equality holds as a sequence $X^n_{i,0}$ has zero probability if it contains a transition of probability zero. Now from Lemma 4.3.1 it follows that

$$\limsup_{n \to \infty} \frac{1}{n} \log \Pr(P^{(2)}_{X^n_{i,0}} \in \bar{R}_i \cap S(W_i) | X_{i,0} = u) \leq - \min_{P \in \Lambda_0 \cap cl(\bar{R}_i \cap S(W_i))} D(P \| W_i).$$

Moreover, since $D(P \| W_i) = \infty$ for any $P \notin S(W_i)$, $\bar{R}_i \cap S(W_i)$ is clearly nonempty, and $\bar{R}_i$ is closed, we have

$$\min_{P \in \Lambda_0 \cap cl(\bar{R}_i \cap S(W_i))} D(P \| W_i) = \min_{P \in \Lambda_0 \cap \bar{R}_i} D(P \| W_i).$$

This yields

$$\limsup_{n \to \infty} \frac{1}{n} \log \Pr(t \neq \hat{t}_n) \leq - \min_i \min_{P \in \Lambda_0 \cap \bar{R}_i} D(P \| W_i).$$

Finally, it remains to show that the right hand side in the above equation is nonzero. Assuming the contrary, we have that for a given i, P_i minimizes $D(P \| W_i)$

in $\Lambda_0 \cap \bar{R}_i$, and $D(P_i \| W_i) = 0$. The latter divergence can be rewritten as

$$D(P_i \| W_i) = \sum_{u:\bar{P}_i(u)>0} \bar{P}_i(u) \sum_v P_i(v|u) \log \frac{P_i(v|u)}{W_i(v|u)} = \sum_{u:\bar{P}_i(u)>0} \bar{P}_i(u) D(P_i(\cdot|u) \| W_i(\cdot|u))$$

where $D(P_i(v|u) \| W_i(v|u))$ denotes the divergence between the two conditional distributions for a given u. Now since $D(P_i \| W_i) = 0$, we have $D(P_i(\cdot|u) \| W_i(\cdot|u)) = 0$ and hence $P_i(v|u) = W_i(v|u)$ for all u with $\bar{P}_i(u) > 0$. Moreover, the irreducibility of W_i implies that if there is a u with $\bar{P}_i(u) = 0$, then there is a u' such that $\bar{P}_i(u') > 0$ and $P_i(u|u') > 0$. Thus, $P_i(u', u) > 0$. Since $P_i \in \Lambda_0$, its marginals are equal, and so $\bar{P}_i(u) = \sum_{\hat{u}} P_i(\hat{u}, u) > 0$, a contradiction. Therefore, $\bar{P}_i(u) > 0$ for all $u \in A$, and so $P_i(v|u) = W_i(v|u)$ for all $u, v \in A$. Moreover, as $P_i \notin R_i$, there is some $j \neq i$ such that $D(P_i \| W_j) \leq D(P_i \| W_i) = 0$. Thus $D(P_i \| W_j) = 0$ by the non-negativity of the relative entropy. Then, since W_j is also irreducible, $P_i(v|u) = W_j(v|u)$ for all $u, v \in A$. Therefore, $W_i = W_j$, a contradiction. □

Remark. Note that the end of the proof heavily depends on the fact that the Markov chains are irreducible. Indeed, it is easy to construct reducible Markov chains such that it is impossible to distinguish between them with vanishing error probability no matter how large the sample size is. For example, consider the following two transition matrices:

$$W_1 = \begin{pmatrix} .5 & .5 & 0 & 0 \\ .5 & .5 & 0 & 0 \\ 0 & 0 & .5 & .5 \\ 0 & 0 & .5 & .5 \end{pmatrix} \quad \text{and} \quad W_2 = \begin{pmatrix} .5 & .5 & 0 & 0 \\ .5 & .5 & 0 & 0 \\ 0 & 0 & .1 & .9 \\ 0 & 0 & .1 & .9 \end{pmatrix}.$$

Then, if the two chains start from state 3 or 4, then it is possible to distinguish between them; however, if they start from state 1 or 2, then the resulting distributions are the same.

Now we are ready to show that $\bar{D}_m$ decays exponentially.

Theorem 4.3.1 *For every block-stationary block-Markov source X_0^∞ there is a constant $c > 0$ depending on the transition matrix of the source such that*

$$\limsup_{m\to\infty} \frac{1}{m} \log \bar{D}_m \le -c.$$

Proof. First notice that $U_0^\infty = \{U_{2kN}^{2(k+1)N-1}\}_{k=0}^\infty$ is a block-$2N$-Markov source for each value of τ, as $U_{2kN}^{2(k+1)N-1}$ always contains a full character of the block-N-Markov source $\{X_{jN}^{(j+1)N-1}\}_{j=0}^\infty$. This fact will enable us to use tools for classification of Markov chains (namely, Lemma 4.3.2) to examine estimates of τ based on the sequence U_0^{m-1}, which then can be used to estimate $\bar{D}_m = I(\tau; U_m|U_0^{m-1})$.

For $\tau = t$, $t \in \{0, \dots, N-1\}$ and $U_0^{2N-1} = w \in A^{2N}$, let $\mathcal{I}_{t,w} \subset A^{2N}$ denote the (irreducible) set of states reachable from w by the Markov chain. Moreover, let $Q_{t,w} = \{q_t(v|u)\}, u, v \in \mathcal{I}_{t,w}$ denote the transition probability matrix corresponding to the states in $\mathcal{I}_{t,w}$. That is,

$$q_t(u|v) = P_{U_{2N}^{4N-1}|U_0^{2N},\tau}(u|v,t) = P_{X_{2N+t-m}^{4N+t-m-1}|X_{t-m}^{2N+t-m-1}}(v|u).$$

To simplify further notation, we extend the above definition for any integer t. Note that $Q_{t,w}$ is a sub-matrix of the transition probability matrix $\{q_t(v|u)\}, u, v \in A^{2N}$ describing the behavior of the Markov chain for all states. Moreover, $Q_{t,w}$ is irreducible for all w, but the index set $\mathcal{I}_{t,w}$ is not necessarily the same for different values of t. If $\mathcal{I}_{t,w} \ne \mathcal{I}_{t',w}$, then the corresponding matrices $Q_{t,w}$ and $Q_{t',w}$ are different. However, $Q_{kN+t,w} = Q_{t,w}$ for any integer k, $t \in \{0, \dots, N-1\}$ and $w \in A^{2N}$, since $X_{-\infty}^\infty$ is block-N-stationary.

In the proof we will try to estimate which $Q_{t,w}$ is the generator matrix of an observed sequence U_0^m. Obviously, if the $Q_{t,w}$ are not all different, this is not possible (we cannot distinguish between two Markov-chains with the same transition matrices). Therefore, for any t let $g_w(t)$ denote the smallest number in $\{0, \ldots, N-1\}$ such that $Q_{t,w} = Q_{g_w(t),w}$, and let N_w^* be the number of different transition matrices $Q_{t,w}$. (It is easy to show that $N_w^* = \max_{0 \le t < N} g_w(t) + 1$, and $Q_{0,w}, \ldots, Q_{N_w^*-1,w}$ are different.)

Moreover, given $U_0^{2N-1} = w$ and $g_w(\tau)$, τ is independent of U_0^m for every m. Therefore,

$$\begin{aligned} & H(\tau | U_{2N}^m, U_0^{2N-1} = w) \\ &\quad = H(\tau, g_w(\tau) | U_{2N}^m, U_0^{2N-1} = w) \\ &\quad = H(g_w(\tau) | U_{2N}^m, U_0^{2N-1} = w) + H(\tau | g_w(\tau), U_{2N}^m, U_0^{2N-1} = w) \\ &\quad = H(g_w(\tau) | U_{2N}^m, U_0^{2N-1} = w) + H(\tau | g_w(\tau)). \end{aligned}$$

By Theorem 4.2.1, this implies for all $m \ge 2N$

$$\begin{aligned} \bar{D}_m &= I(\tau; U_m | U_0^{m-1}) = H(\tau | U_0^{m-1}) - H(\tau | U_0^m) \\ &= \sum_w P_{U_0^{2N-1}}(w) \left(H(g_w(\tau) | U_{2N}^{m-1}, U_0^{2N-1} = w) - H(g_w(\tau) | U_{2N}^m, U_0^{2N-1} = w) \right) \\ &\le \sum_w P_{U_0^{2N-1}}(w) H(g_w(\tau) | U_{2N}^{m-1}, U_0^{2N-1} = w) \\ &\le \max_{w : P_{U_0^{2N-1}}(w) > 0} H(g_w(\tau) | U_{2N}^{m-1}, U_0^{2N-1} = w). \end{aligned}$$

Therefore,

$$\limsup_{m\to\infty} \frac{1}{m} \log \bar{D}_m \le \max_{w : P_{U_0^{2N-1}}(w) > 0} \limsup_{m\to\infty} \frac{1}{m} \log H(g_w(\tau) | U_{2N}^{m-1}, U_0^{2N-1} = w). \quad (4.1)$$

Next we bound the conditional entropies $H(g_w(\tau) | U_{2N}^{m-1}, U_0^{2N-1} = w)$. If $N_w^* = 1$, then $g_w(\tau) = 0$ with probability 1, and so $H(g_w(\tau)|U_{2N}^{m-1}, U_0^{2N-1} = w) = 0$ and

$$\lim_{m\to\infty} \frac{1}{m} \log H(g_w(\tau)|U_{2N}^{m-1}, U_0^{2N-1} = w) \le -c_w \tag{4.2}$$

with $c_w = \infty$. Otherwise, if $N_w^* > 1$, let $\tau_{m,w} = \tau_{m,w}(U_0^{m-1})$ be an optimal estimate of $g_w(\tau)$ based on U_0^{m-1}, given $U_0^{2N-1} = w$ in the sense that $\Pr(g_w(\tau) = \tau_{m,w}) \ge \Pr(g_w(\tau) = f(U_0^{m-1})|U_0^{2N-1} = w)$ for any function $f : A^m \to \{0, \dots, N^* - 1\}$, and let

$$p_{m,w} = \Pr(g_w(\tau) \ne \tau_{m,w}|U_0^{2N-1} = w)$$

(note that such an estimate always exist). Then, since $\tau_{m,w}$ is a function of U_0^m,

$$H(g_w(\tau)|U_{2N}^{m-1}, U_0^{2N-1} = w) \le H(g_w(\tau)|\tau_{m,w}, U_0^{2N-1} = w) \tag{4.3}$$

(for properties of the entropy function see, e.g., [8]). Moreover, by Fano's inequality

$$H(\tau^*|\tau_m, U_0^{2N-1} = w) \le \log(N_w^* - 1)p_{m,w} + h_b(p_{m,w})$$

where $h_b(p) = -p \log p - (1-p)\log(1-p)$ for $0 \le p \le 1$. From here, obviously

$$\begin{aligned} &\limsup_{m\to\infty} \frac{1}{m} \log H(g_w(\tau)|\tau_{m,w}, U_0^{2N-1} = w) \\ &\le \limsup_{m\to\infty} \frac{1}{m} \log \left(2 \max\{\log(N_w^* - 1)p_{m,w}, h_b(p_{m,w})\}\right) \\ &\le \max\left\{\limsup_{m\to\infty} \frac{1}{m} \log p_{m,w},\ \limsup_{m\to\infty} \frac{1}{m} \log h_b(p_{m,w})\right\}. \end{aligned} \tag{4.4}$$

Next we use Lemma 4.3.2 to bound (4.4). In order to be able to apply the lemma, we need to determine the state space of the observed process, and then we only need to find the generating Markov chain (given by $Q_{t,w}$) among those that live on that state space. Since the Markov chains defined by the matrices $Q_{t,w}$ are irreducible, the probability that a given state is not reached in k steps converges to 0 exponentially fast in k. Therefore, for the set of values $\hat{\mathcal{I}}_k = \{U_0^{2N-1}, U_{2N}^{4N-1}, \ldots, U_{2(k-1)N}^{2kN-1}\}$, we have

$$\lim_{k\to\infty} \Pr(\hat{\mathcal{I}}_k \neq \mathcal{I}_{t,w} | \tau = t, U_0^{2N-1} = w) \leq -c'_{w,t}$$

for some $c'_{w,t} > 0$. This implies that

$$\lim_{m\to\infty} \Pr(\hat{\mathcal{I}}_{\lfloor \frac{m}{2N} \rfloor} \neq \mathcal{I}_{g_w(\tau)} | U_0^{2N-1} = w) \leq -c_{w,1} \tag{4.5}$$

where $c_{w,1} = \min_t c'_{w,t}/2N > 0$.

For any $\mathcal{I} \subset A^{2N}$, let

$$\mathcal{Q}_w(\mathcal{I}) = \{g_w(i) : 0 \leq i < N,\ Q_{i,w} \text{ is defined on } \mathcal{I}\}$$

denote the set of indexes of the Markov chains with state space $\mathcal{I}$ (for Markov chains with the same transition matrix, we consider the one with the smallest index). Now $g_w(\tau)$ can be estimated by first estimating $\mathcal{I}_{g_w(\tau)}$ by $\hat{\mathcal{I}}_k$, $k = \lfloor (m-1)/2N \rfloor$, based on U_0^m, and then estimating $g_w(\tau)$ by the optimal classifier for the problem of deciding which $Q_{i,w}, i \in \mathcal{Q}_w(\hat{\mathcal{I}}_k)$ generated the sequence $U_0^{2N-1} = w, U_{2N}^{4N-1}, \ldots, U_{2(k-1)N}^{2kN-1}$. Let $p'_{m,w}$ denote the conditional error probability of the optimal classifier $\hat{\mathcal{I}}_k$ for $\mathcal{I}_{g_w(\tau)}$ given $U_0^{2N-1} = w$. Then

$$p_{m,w} \leq \Pr(\hat{\mathcal{I}}_k \neq \mathcal{I}_{g_w(\tau)} | U_0^{2N-1} = w) + p'_{m,w}. \tag{4.6}$$

Let

$$R_i = \{P \in \Lambda : D(P \| Q_{i,w}) < D(P \| Q_{j,w}) \text{ for all } i \neq j,\ i, j \in \mathcal{Q}_w(\mathcal{I}_{g_w(\tau)})\}$$

and define

$$c_{w,2} = \min_{i \in \mathcal{Q}_w(\mathcal{I}_{g_w(\tau)})} \min_{P \in \Lambda_0 \cap \bar{R}_i} D(P \| Q_{i,w}).$$

Then, as the $\{Q_i\}, i \in \mathcal{Q}_w(\mathcal{I}_{g_w(\tau)})$ are different and irreducible, from Lemma 4.3.2 we have $c_{w,2} > 0$ and

$$\limsup_{m \to \infty} \frac{1}{m} \log p'_{m,w} \leq -c_{w,2}.$$

Combining this inequality with (4.5) and (4.6) we obtain that for the positive number $c_w = \min\{c_{w,1}, c_{w,2}\}$ we have

$$\limsup_{m \to \infty} \frac{1}{m} \log p_{m,w} \leq -c_w. \tag{4.7}$$

In particular, $\lim_{m \to \infty} p_{m,w} = 0$. Therefore, as L'Hospital's rule implies

$$\lim_{p \to 0} p \log(1/p)/h_b(p) = 1$$

we have

$$\begin{aligned} \limsup_{m \to \infty} \frac{1}{m} \log h_b(p_{m,w}) &= \limsup_{m \to \infty} \frac{1}{m} \log \left(p_{m,w} \log \frac{1}{p_{m,w}} \right) \\ &= \limsup_{m \to \infty} \frac{1}{m} \log p_{m,w} \end{aligned}$$

where the second equality holds because (4.7) implies $\lim_{m \to \infty} \frac{1}{m} \log \log p_{m,w} = 0$. Thus, the two terms in the maximum in (4.4) are equal and converge to zero exponentially fast by (4.7). Combining this fact with inequalities (4.1) and (4.2) proves the theorem. $\square$

4.4 Universal symbol-based coding of block Markov sources

Now we are ready to establish an upper bound for the real coding redundancy for a large class of universal symbol-based codes. Let $\ell_n^{(m)} : A^n \to \mathbb{N}^+$ denote the code lengths of a universal code $\{f_n\}$ for mth order Markov sources satisfying

$$\frac{1}{n} \sup_{P_{Y_0^{n-1}}} \sup_{z_0^{n-1} \in A^n} \left[\ell_n^{(m)}(z_0^{n-1}) + \log P_{Y_0^{n-1}}(z_0^{n-1}) \right] \le c_n^{(m)} \tag{4.8}$$

for some $c_n^{(m)} \to 0$ as $n \to \infty$, where the first supremum is taken over all n-fold marginal distributions of mth order Markov sources over A. In other words, we require that the "pointwise redundancy" converges to zero uniformly for each source sequence and for each mth order Markov source. For example, there exist universal arithmetic codes for mth order Markov sources with $c_n^{(m)} = O(|A|^{m+1} \log n / n)$ (see, e.g., [11]).

For fixed m and n the per symbol coding redundancy is defined as

$$R_{n,m} = \frac{1}{n} \left(E\ell_n^{(m)}(X_0^{n-1}) - H(X_0^{n-1}) \right)$$

Theorem 4.4.1 *If the code length function $\ell_n^{(m)}$ satisfies (4.8) then for $n \ge m \ge 2N$ the coding redundancy $R_{n,m}$ for the block-stationary block-Markov source X_0^∞ can be bounded as*

$$R_{n,m} \le \frac{1}{n} \log N + 2^{-mc_r + o(m)} + c_n^{(m)} \tag{4.9}$$

where c_r is defined in Theorem 4.3.1.

Remarks.

(i) For any fixed m and very large coding block length n, the redundancy is exponentially small in m, that is,

$$\limsup_{n\to\infty} \frac{1}{n} E[\ell_n^{(m)}(X_0^{n-1})] - \bar{H}(X_0^\infty) \leq 2^{-mc_r+o(m)}.$$

(ii) It is easy to see that to minimize the bound (4.9), m should be chosen $O(\log n)$. As mentioned before, there are arithmetic codes with $c_n^{(m)} = O(|A|^{m+1}\log n/n)$ [11]. For these codes, the optimal choice is $m = \log n/(c_r + \log|A|)$, yielding

$$R_{m,n} = O\left(n^{-\frac{c_r}{c_r+\log|A|}}\right).$$

Obviously, c_r is not known in advance. Moreover, this rate is slower than applying the universal code to the first order block Markov source, which results in $O(N|A|^{2N}\log(n/N)/n)$ redundancy. The reason for this is that while the number of parameters of the original source is finite (namely, $|A|^{2N}$), the number of parameters of the approximating mth order Markov chain (which is $|A|^m$) grows without bound as m increases. On the other hand, if the dependence of $c_n^{(m)}$ on m is less than exponential, then the dominant term in (4.9) is usually the last one.
(iii) The result is significant for the practical case of universal compression, when the block size of the input is not known. Choosing an incorrect block length may result in deteriorated performance, as shown by the following experiment. We compressed the English-language text "book1" from the Calgary Corpus [42] represented as a binary sequence with blocks of fixed length ($N_s = 7$ and $N_s = 8$) corresponding to characters, using the `bzip2` algorithm operating on (possibly different) fixed-length blocks of N_e symbols (N_e is chosen to be 1,7, and 8). Obviously, the plots when $N_s = N_e$ are the same. The per-block entropy rate of the source does not depend on N_s, equalling approximately 2 bits per block. The graph on Figure 4.4 shows the average number of bits in the encodings per one source block, as the length of the source sequence (measured in source blocks) increases.

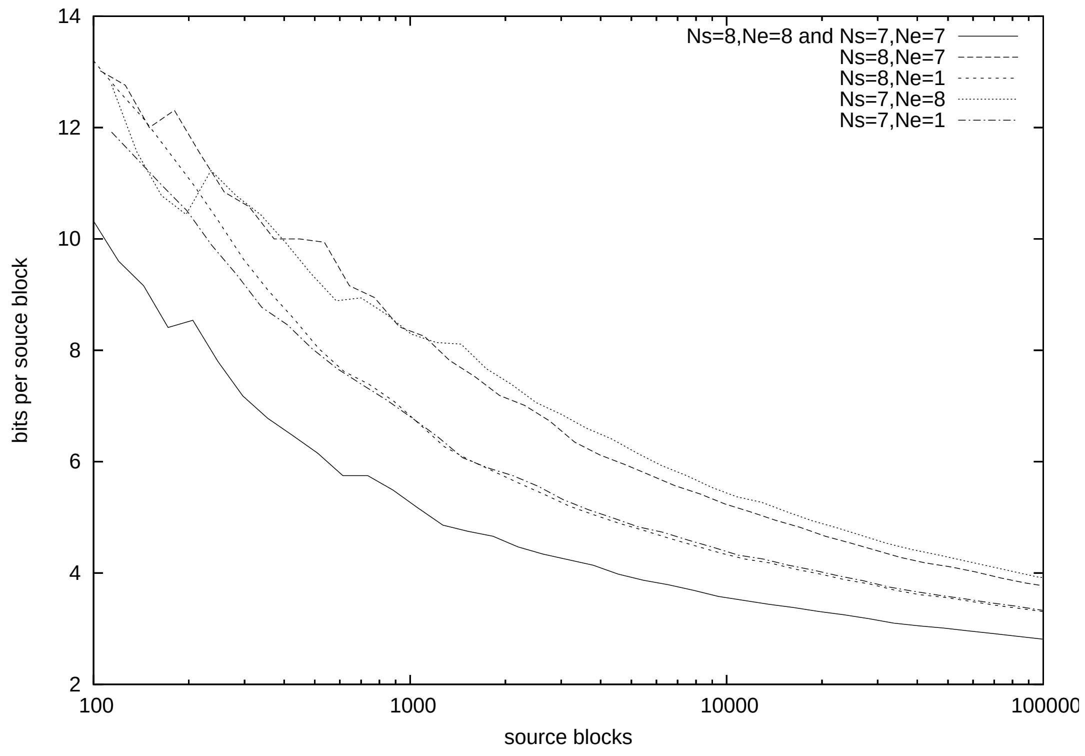

Figure 4.1: Average number of bits per source block

Choosing the smallest encoding block length $N_e = 1$ results in guaranteed performance by Theorem 4.4.1, with the computational advantage of operating on a small alphabet. Thus, coding on the elementary symbol level is a practically good suboptimal scheme for encoding block Markov sources with unknown block size.

Proof of Theorem 4.4.1: From (4.8) it follows that for any mth order Markov source Y_0^∞ and x_0^{n-1}

$$\log \frac{P_{Y_0^{n-1}}(x_0^{n-1})}{P_{\ell_n^{(m)}}(x_0^{n-1})} \le nc_n^{(m)} \tag{4.10}$$

where $P_{\ell_n^{(m)}}$ denotes the coding distribution for n-long sequences.

Let $\widehat{Y}_0^\infty$ denote the stationary mth order Markov approximation of X_0^∞, defined in Theorem 4.2.1, achieving the minimum in the definition of $\bar{D}_m$ (recall that $P_{\widehat{Y}_0^{m-1}} =$

$P_{U_0^{m-1}})$. Then

$$\begin{aligned} & D(P_{X_0^{n-1}} \| P_{\ell_n^{(m)}}) \\ = \;& D(X_0^{n-1} \| \widehat{Y}_0^{n-1}) + \sum_{z_0^{n-1} \in A^n} P_{X_0^{n-1}}(z_0^{n-1}) \log \frac{P_{\widehat{Y}_0^{n-1}}(z_0^{n-1})}{P_{\ell_n^{(m)}}(z_0^{n-1})} \\ \leq \;& D(X_0^{n-1} \| \widehat{Y}_0^{n-1}) + n c_n^{(m)} \end{aligned}$$

where the inequality holds by (4.10). Now, the first term can be easily bounded following the proof of Theorem 4.2.1 as

$$\begin{aligned} D(X_0^{n-1} \| \widehat{Y}_0^{n-1}) &= D(X_0^{m-1} \| \widehat{Y}_0^{m-1}) + \sum_{i=m}^{n} D(X_i | X_0^{i-1} \| Y_i | Y_0^{i-1}) \\ &\leq D(X_0^{m-1} \| \widehat{Y}_0^{m-1}) + \sum_{i=m}^{m-1+N\left\lceil \frac{n-m+1}{N} \right\rceil} D(X_i | X_0^{i-1} \| Y_i | Y_0^{i-1}) \\ &= D(X_0^{m-1} \| \widehat{Y}_0^{m-1}) + \left\lceil \frac{n-m+1}{N} \right\rceil \sum_{t=0}^{N-1} S_t \\ &\leq D(X_0^{m-1} \| \widehat{Y}_0^{m-1}) + n\bar{D}_m \end{aligned}$$

where S_t is defined as in the proof of Theorem 4.2.1 with $Y_0^\infty = \widehat{Y}_0^\infty$. Furthermore,

$$D(X_0^{m-1} \| \widehat{Y}_0^{m-1}) = D(X_0^{m-1} \| U_0^{m-1}) \leq \log N$$

since for any $x_0^{m-1} \in A^m$, $P_{U_0^{m-1}}(x_0^{m-1}) \geq P_{X_0^{m-1}}(x_0^{m-1})/N$ by definition. Thus, by Theorem 4.3.1

$$R_{n,m} \leq \frac{1}{n} \log N + 2^{-mc_r + o(m)} + c_n^{(m)}.$$

□

4.5 Conclusion

We have demonstrated that block-Markov sources can be encoded with exponentially fast vanishing redundancy using codes that are optimized for higher-order symbol-level Markov models. This partially explains the findings of our experiments (see Chapter 3) that a bit-level implementation of a universal compression algorithm performs reasonably well on byte-aligned data when compared with byte-level implementations, inviting further studies of bit-level implementations of compression algorithms, as on the bit level, one can take advantage of the computational benefits of operating on the smallest possible alphabet, as demonstrated in the next chapter.

Chapter 5

Algorithmic Results

The algorithms presented and analyzed in this chapter have been developed for the purposes of efficient block-sorting compression, but they are also useful for other source coding and string processing purposes. The main result here is a generalization of Ukkonen's algorithm, introduced in Section 2.10.3, for blocks of symbols rather than individual symbols, allowing for large alphabets without relinquishing the advantages of small ones.

In particular, an efficient representation of binary suffix trees is presented followed by a generalization for representing suffix trees of strings of symbols that can be represented as blocks of bits (e.g. bytes). Upper bounds on memory and time costs are proved.

5.1 Introduction

The memory footprint of the suffix tree, albeit linear in the length of the string (denoted by n), is rather large. The constant factor depends on the actual representation of nodes and arcs and the alphabet. In case of a binary alphabet with no string termination and arcs represented by 32-bit pointers, we have 160 bits for each explicit node (5 pointers: two for the children, two for *beginning* and *end* and one for the suffix link), which is an almost 320-fold overhead over the original string. By introducing the end-of-string symbol, another 32 bits are added to each node. If the alphabet is much larger (e.g., that of bytes) representing arcs by indexed pointers becomes impractical; in this case, arc representation requires a sophisticated data structure of its own (in practice, hash tables are used most often). An interesting exception is [16], where indexed pointers are used for large alphabets, but in that case one runs into dynamic memory management problems as the pointer arrays have different lengths.

In this chapter, an alternative representation is proposed where the symbols of the large alphabet are replaced by blocks of elementary symbols from a small alphabet (e.g., bits), while the low number of explicit nodes is retained. The end-of-string symbol is represented implicitly. In case of 8-bit blocks and 32-bit pointers, this representation results in a less than 40-fold overhead in the worst case.

5.2 Preliminaries

So far, we have paid little attention to the actual representations of the various objects and references of which the suffix tree described in Section 2.10.2 consists and of those that are used in actually constructing it (see Section 2.10.3). In this chapter, we will carefully examine these data structures at a lower level of abstraction.

A possible straightforward representation of explicit nodes, suffix links and arcs of suffix tree $S(t_0^{n-1})$ would be one consisting of building blocks as depicted by Table 5.1.

For a particular explicit node corresponding to string $\mathbf{s}$, we denote the index of the first symbol of the label of the incoming arc by $\mathbf{s} \rightarrow \mathit{start}$, the index of the symbol following the last symbol of the label by $\mathbf{s} \rightarrow \mathit{end}$ and the explicit node pointed by the suffix link (that is the one corresponding to $s_1^{|\mathbf{s}|-1}$) by $\mathbf{s} \rightarrow \mathit{link}$. The first explicit node corresponding to or following $\mathbf{s}x$, if such a node exists, is denoted by $\mathbf{s} \rightarrow \mathit{child}[x]$, where $x \in A$ is the first symbol in the label of the incoming arc of said explicit node.

field	**type**
start index	symbol index such that $0 \le start \le end$
end index	symbol index such that $start \le end \le n$
suffix *link*	pointer to another explicit node
child pointers	$\lvert A \rvert$ pointers to other explicit nodes

Table 5.1: Direct representation of explicit nodes in a suffix tree

Thus, the difference between the *end* and *start* indices is the length of the label of the incoming arc and the explicit nodes at the other end of outgoing arcs are pointed by the *child* pointers.

This representation is quite efficient for the binary alphabet if we disregard for the moment the need to represent the end of the string. As described in the next section, that can be solved without modifying this basic data structure. For a larger alphabet A' (with $|A'| \gg 2$), it can be rather wasteful, since most *child* pointers are not used. Instead of using a different data structure, however, we treat the symbols of A' as blocks of $N = \lceil \log |A'| \rceil$ bits.

Remark: To save further space, leaf nodes may be represented without the *child* pointers, with the *end* pointer pointing to the end of the string, thus conveying the leaf type of the node.

Now we are ready to define the *block-N suffix tree* of t_0^n, and other auxiliary data structures. The following definitions are direct generalizations of Definition 2.10.1 and Definition 2.10.2.

Definition 5.2.1 (Block-N Suffix Trie) *Let $V_N(t_0^{n-1})$ denote the set of different substrings of t_0^{n-1} beginning at indices divisible by N. The directed graph $T_N(t_0^{n-1})$ over the vertices $V_N(t_0^{n-1})$ is called the* block-N suffix trie *of t_0^{n-1} if its arc set $E_N(t_0^{n-1})$*

satisfies

$$\overrightarrow{\mathbf{x}|\mathbf{y}} \in E_N(t_0^{n-1}) \text{ iff } t_{iN}^{iN+|\mathbf{x}|} = \mathbf{x}t_{iN+|\mathbf{x}|} = \mathbf{y} \text{ for some } i \in \mathbb{N}$$

Definition 5.2.2 (Block-N Suffix Tree) $S_N(t_0^{n-1}) = (W_N(t_0^{n-1}, F_N(t_0^{n-1}))$*, the* block-$N$ suffix tree *of* t_0^{n-1} *is defined by contracting all arcs of* $T_N(t_0^{n-1})$ *which are the sole outbound arc of a node.*

For any node corresponding to $\mathbf{s}$, the suffix link $\mathbf{s} \rightarrow$ *link* in these cases points to the node corresponding to $s_N^{|\mathbf{s}|-1}$, if $N < |\mathbf{s}|$. In case of equality, it points to the node corresponding to ϵ denoted henceforth as the *root*.

Note that the suffix trie and the suffix tree of Definition 2.10.1 and Definition 2.10.2 are special cases of the block suffix trie and the block suffix tree, respectively, for the case $N = 1$. Formally, $T(t_0^{n-1}) = T_1(t_0^{n-1})$ and $S(t_0^{n-1}) = S_1(t_0^{n-1})$.

The following definitions from [39] can be applied to the generalized data structures essentially without modification:

Definition 5.2.3 (Reference Pair) *Nodes* $\mathbf{u} \in V_N(t_0^{n-1})$ *of* $T_N(t_0^{n-1})$ *(both explicit and implicit) are referenced by the* reference pair $(\mathbf{s}, \mathbf{w})$ *where* $\mathbf{s} \in W_N(t_0^{n-1})$ *is an explicit node and* $\mathbf{w}$ *is a substring of* t_0^{n-1} *such that* $\mathbf{u} = \mathbf{sw}$.

On the implementation level, reference pairs are represented by a pointer $\bar{s}$ to the data structure (see Table 5.1) corresponding to explicit node $\mathbf{s} \rightarrow$ *child*$[w_0]$ and an index c and length l such that $\mathbf{w} = t_c^{c+l-1}$.

The reference pair to some particular $\mathbf{u} \in V_N(t_0^{n-1})$ is obviously not unique. Of the many possibilities, we define the *canonical reference pair* to $\mathbf{u}$ as follows:

Definition 5.2.4 (Canonical Reference Pair) *A reference pair* $(\mathbf{s}, \mathbf{w})$ *is* canonical *if* $\mathbf{s}$ *is the closest ancestor of* $\mathbf{u}$ *in* $T_N(t_0^{n-1})$.

The following algorithm makes a reference pair canonical, if it is not canonical already:

Algorithm 5.2.1 (Canonize)
Input: $\bar{s}$, c, l
Output: $\bar{s}$, c, l

```
1. $skip \leftarrow (\bar{s} \rightarrow end - \bar{s} \rightarrow start)$
2. if $l < skip$ then return
3. $c \leftarrow (c + skip)$
4. $\bar{s} \leftarrow (\bar{s} \rightarrow child[t_c])$
5. $l \leftarrow (l - skip)$
6. go to 1
```

In fact, Algorithm 5.2.1 is not different from procedure *canonize* in [39, Section 4]. Its apparent simplicity is the result of the underlying representation.

5.3 End of String Representation

In many applications, including the Burrows-Wheeler transformation, there is an additional unique end-of-string symbol $t_n \notin A$ added to the string t_0^{n-1} so that there is a one-to-one correspondence between leaves of $S(t_0^n)$ and suffixes of t_0^{n-1}. For example, in this case the number of occurrence of a substring is equal to the number of leaf successors of its node. That is in this case, the number of leaves of $S(u_0^{n-1})$ (and that of isomorphic representations) always equals $n + 1$.

If the suffix tree on a larger alphabet is represented as a *block-suffix tree* on bits as suggested in the next section, explicit end-of-string representation may become very expensive. In case of using a third symbol besides one and zero, an additional pointer must be added to each explicit node in order to allow for three-way branchings. If the end-of-string symbol is introduced on the level of supersymbols (blocks of bits), and $|A| = 2^N$ as is often the case, with the introduction of a new symbol, one must add an additional bit to the representation of the supersymbols. This, in addition to leading to a substantial size increase, can eliminate the possibility of block-operations on the computer (e.g., by turning bytes into 9-bit blocks).

The following two observations allow for an implicit representation of the unique end-of-string character, regardless of whether the suffix tree is represented using its native symbols or as a block-suffix tree on elementary symbols. The first is simply a special case of the second, with block length $N = 1$.

Lemma 5.3.1 *In a block-suffix tree $S_N(t_0^{nN})$, where $t_{nN} \neq t_i$ for every $i \neq nN$, the labels of arcs to explicit nodes having a child labeled by t_{nN} are suffixes of t_0^{nN-1}.*

Lemma 5.3.2 *In a block-suffix tree $S_N(t_0^{nN})$, where $t_{nN} \neq t_i$ for every $i \neq nN$, there are no* end *pointers with index nN, except those of explicit nodes having a child labeled by t_{nN}.*

Lemma 5.3.1 follows directly from the definition of explicit node labels, while Lemma 5.3.2 follows from the on-line nature of Ukkonen's algorithm, assigning *end* pointers in increasing order. Therefore, before t_{nN} is added, there are no *end* pointers with index nN.

Lemma 5.3.3 *It is possible to modify $S_N(t_0^{nN})$ so that those and only those explicit nodes have an* end *pointer pointing to t_{nN} that have a child labeled by it.*

Proof: From Lemma 5.3.1, it follows that it is possible to modify S_{sN} so that all the explicit nodes having a child labeled by t_{nN} have their *end* pointer pointing to t_{nN}. Lemma 5.3.2 ensures that after doing so, no other explicit node will have an *end* pointer with index nN. □

Observe that the leaves labeled by t_{nN} can be deleted from the block suffix tree modified according to Lemma 5.3.3 without loss of information, as the *end* pointers uniquely identify explicit nodes to which the deleted leaves were attached. Similarly, t_{nN} can be deleted from the labels of the remaining leaves by decreasing their *end* pointer's index from $nN + 1$ to nN. Note, furthermore, that these explicit nodes are the ones on the boundary path of $S_N(t_0^{nN-1})$, which is a subgraph of $S_N(t_0^{nN})$.

By doing so, we obtain a representation of $S_N(t_0^{nN})$, with no explicit end-of-string symbols. The following theorem states that this same representation — denoted by $S'_N(t_0^{nN})$ — can be obtained from $S_N(t_0^{nN-1})$ with a time cost proportional to n.

Theorem 5.3.1 $S'_N(t_0^{nN})$ *can be constructed from* $S_N(t_0^{nN-1})$ *using* $O(n)$ *operations.*

Proof: The boundary path of $S_N(t_0^{nN-1})$ has $n+1$ nodes on it. By traversing them, implicit nodes along the boundary path are first made explicit (by splitting the node into two), and for all (now explicit) nodes along the boundary path the *beginning* and *end* pointers are incremented by the same amount so that the *end* index equals nN. The result is $S'_N(t_0^{nN})$. $\square$

In practice, one can begin the traversal of the boundary path at the so-called *active point* of $S_N(t_0^{nN-1})$, which is the first node along the boundary path that is not a leaf. The active point is known after the execution of Ukkonen's algorithm.

5.4 Block Suffix Trees

A block-N suffix tree $S_N(t_0^{n-1})$ of t_0^{n-1} represents a tree graph over all substrings t_{iN}^{iN+l}, where $0 \leq i < n/N$and $0 \leq l < n - iN$ with directed arcs $\overrightarrow{t_{iN}^{iN+k}|t_{jN}^{jN+k+1}}$ labeled by t_{jN+k+1} for every $t_{iN}^{iN+k} = t_{jN}^{jN+k}$, as previously. Suffix links are defined from t_{iN}^{iN+k} to t_{iN+N}^{iN+k} if $k > N$. Note that in this definition no suffix link originates from nodes corresponding to substrings not longer than N. Nodes with only one outbound arc are contracted as in the case of suffix trees.

Similarly to Ukkonen's auxiliary state $\perp$, one can introduce additional N nodes denoted by $\lambda_0 \ldots \lambda_{N-1}$ above the *root* in order to avoid a distinction between nodes corresponding to short substrings and those consisting of more than N symbols. In actual implementations, these do not need to be represented explicitly. Suffix links to *root* point from all nodes corresponding to substrings of length N, while from those of length $k < N$ the suffix link points to λ_k. The suffix link from *root* points to λ_0. arcs labeled by all symbols of A connect λ_i to λ_{i+1} and λ_{N-1} to *root*.

For a string t_0^{nN-1} on alphabet A and u_0^{n-1} on alphabet A^N where $u_i = t_{iN}^{iN+N-1}$, the one-to-one mapping $\phi : u_i^{i+k} \to t_{iN}^{(i+k)N}$ from the nodes of the suffix tree $S(u_0^{n-1})$ to a subset of the nodes of block-suffix tree $S_N(t_0^{nN-1})$ preserves connectivity; thus $S_N(t_0^{nN-1})$ can be considered as a representation of $S(u_0^{n-1})$:

Lemma 5.4.1 *The node corresponding to u_i^{i+k} is the direct ancestor of that corresponding to u_j^{j+k+1} in $S(u_0^{n-1})$ if and only if $\phi(u_i^{i+k})$ is the Nth ancestor of $\phi(u_j^{j+k+1})$ in $S_N(t_0^{nN-1})$.*

Proof: $u_i^{i+k} = u_j^{j+k}$ iff $t_{iN}^{iN+kN+m} = t_{jN}^{jN+kN+m}$ for every $m \in \{0, \dots, N-1\}$, thus t_{iN}^{iN+kN} is the direct ancestor of $t_{iN}^{iN+kN+1}$, which is the direct ancestor of $t_{iN}^{iN+kN+2}$and so on, up to $t_{iN}^{iN+kN+N-1}$, which is the direct ancestor of $t_{jN}^{(j+k+1)N}$. □

Lemma 5.4.2 *The end indices of all mth successors of a node t_i^{i+k} in S_N have the same remainder after division by N.*

Proof: They are at the same distance $i + k + m$ from *root*, and the beginning indices of the corresponding substrings are divisible by N. □

Lemma 5.4.1 expresses the essential isomorphism between the two graphs corresponding to $S(u_0^{n-1})$ and $S_N(t_0^{nN-1})$. Similarly, there is an isomorphism between the actual data structures, the concise representations with implicit nodes:

Theorem 5.4.1 *Nodes u_i^{i+k} and u_j^{j+k+1} are contracted in $S(u_0^{n-1})$ iff nodes $\phi(u_i^{i+k})$ and $\phi(u_j^{j+k+1})$ are contracted in $S_N(t_0^{nN-1})$. Hence, ϕ defines an isomorphism between $S(u_0^{n-1})$ and $S_N(t_0^{nN-1})$.*

Proof: If nodes $\phi(u_i^{i+k})$ and $\phi(u_j^{j+k+1})$ are contracted, it means that each node on the path $t_{iN}^{(i+k)N+m}$, where $m \in \{0, 1, \dots, N-1\}$ has exactly one successor. Therefore $\phi(u_i^{i+k})$ has exactly one Nth successor, thus, according to Lemma 5.4.1, u_i^{i+k}

and u_1^{j+k+1} are contracted as well (and $i = j$). The other direction can be proved by contradiction. Suppose that $\phi(u_i^{i+k})$ and $\phi(u_j^{j+k+1})$ are not contracted, but u_i^{i+k} and u_j^{j+k+1} are. This, by Lemma 5.4.1, implies that there is at least one branching somewhere on the path between $\phi(u_i^{i+k})$ and $\phi(u_j^{j+k+1})$, but $\phi(u_j^{j+k+1})$ is the sole Nth successor of $\phi(u_i^{i+k})$. This, in turn, implies that it has a leaf successor that is closer than N. But since t_0^{nN-1} has length nN, all leaves are at a distance divisible by N from *root*, which is in contradiction with Lemma 5.4.2. □

The block suffix tree $S_N(t_0^{nN-1})$ has at most n leaves and $n-1$ explicit nodes. While these upper bounds are the same as those for the suffix tree $S(u_0^{n-1})$, it is worth noting that $S_N(t_0^{nN-1})$ has usually more explicit nodes, as more than $|A|$-way branchings in the original suffix tree are represented by a system of at most $|A|$-way branchings. Yet, because of the need for a separate database for the arcs in the direct representation of $S(u_0^{n-1})$, representing it in a block-suffix tree still saves memory and time.

Block suffix trees can be constructed using Ukkonen's algorithm ([39], Algorithm 2) by modifying the initialization steps 1,2 and 3 to create $\lambda_0 \dots \lambda_{N-1}$, the arcs between them and the suffix link from *root* to λ_0 and the arcs from λ_{N-1} to *root*. The rest of the algorithm and its procedures apply without modifications, as well as the proof of linear time cost.

Using the low-level representation from Section 5.2 for the binary alphabet $A = \{0, 1\}$ offers several opportunities for simplification, warranting separate treatment.

First, observe that *an explicit node cannot be the active point.* This renders procedure *test-and-split* in [39, Section 4] trivial. Essentially, in that step, one always makes the active point explicit by splitting the corresponding edge. Secondly, because of the particular low-level representation, there is no need to actually represent $\lambda_0 \dots \lambda_{N-1}$; suffix links pointing to those can be represented by *null* pointers.

For the sake of simplicity, we incorporated procedures *update* and *test-and-split* from [39] in the following step-by-step description of the generalized version of Ukkonen's algorithm for binary block suffix trees. For $N = 1$, it becomes Ukkonen's algorithm. The termination step transforming $S_N(t_0^{n-1})$ into $S_N(t_0^n)$ has not been included in the listing below.

Algorithm 5.4.1 (Binary Block-N Suffix Tree Construction)
Input: t_0^{n-1}, N
Output: $S_N(t_0^{n-1})$

Initialization.

```
1. $l \leftarrow 0, i \leftarrow 0, p \leftarrow N, q \leftarrow N$
2. $from \leftarrow null$
3. add new leaf node $root$ to $S_N$
4. $(root \rightarrow start) \leftarrow 0, (root \rightarrow end) \leftarrow \infty, (root \rightarrow link) \leftarrow null$
5. $\bar{s} \leftarrow root$
```

Main loop until the end of the input string.

```
6. while $p < n$ do
```

Search for the active point. Actually, if the computer can operate on blocks of bits (as is usually the case), this loop can be implemented in a much more efficient way. Here it is presented like this for the sake of clarity.

```
7.         while $i < N$ do
8.                 $x \leftarrow t_{p+i}$
9.                 if $l = (\bar{s} \rightarrow end - \bar{s} \rightarrow start)$ then
10.                        $l \leftarrow 0$
11.                        $\bar{s} \leftarrow (\bar{s} \rightarrow child[x])$
12.                else
13.                        $x \leftarrow t_{p+i}$
14.                        if $t_{\bar{s} \rightarrow start+l} \neq t_{p+i}$ then break to 16
15.                $i \leftarrow (i+1), l \leftarrow (l+1)$
16.        if $i < N$ then do
```

Active point found. The following procedure splits the arc on which the active point resides. The new leaf is added as a *branch*, while the rest of the arc is the *trunk*. This corresponds to *test-and-split* in [39].

```
17.             add new node $trunk$ to $S_N$
18.             $(trunk \rightarrow end) \leftarrow (\bar{s} \rightarrow end) \leftarrow (\bar{s} \rightarrow start + l)$
19.             $(trunk \rightarrow start) \leftarrow (\bar{s} \rightarrow end)$
20.             $(trunk \rightarrow child[0]) \leftarrow (\bar{s} \rightarrow child[0])$
21.             $(trunk \rightarrow child[1]) \leftarrow (\bar{s} \rightarrow child[1])$
22.             add new leaf node $branch$ to $S_N$
23.             $(branch \rightarrow start) \leftarrow (p + i), (branch \rightarrow end) \leftarrow \infty$
24.             $(\bar{s} \rightarrow child[x]) \leftarrow branch, (\bar{s} \rightarrow child[1 - x]) \leftarrow trunk$
```

Traversal of the boundary path. The two branches of the condition below deal with explicit nodes and $\lambda_0 \ldots \lambda_N$, respectively.

```
25.             if $from \neq null$ then $(from \rightarrow link) \leftarrow \bar{s}$
26.             $from \leftarrow \bar{s}$
27.             if $(\bar{s} \rightarrow link) \neq null$ then
28.                 $c \leftarrow (\bar{s} \rightarrow start), \bar{s} \leftarrow (\bar{s} \rightarrow link)$
29.                 canonize
30.                 if $l = 0$ or $t_{\bar{s} \rightarrow start + l} = x$ then $from \leftarrow null$
31.             else if $q < p$ then
32.                 $p \leftarrow (p + N), c \leftarrow p$
33.                 $l \leftarrow (p - q) + ((\bar{s} \rightarrow start + l) \text{mod} N)$
34.                 $\bar{s} \leftarrow root$
35.                 canonize
36.                 $from \leftarrow null$
37.             else
38.                 $\bar{s} \leftarrow root$
39.                 $l \leftarrow 0, i \leftarrow 0, from \leftarrow null$
40.                 $p \leftarrow (p + N), q \leftarrow (q + N)$
41.         loop while $from \neq null$
```

Active point not found in the block. Continue searching.

```
42.         else $q \leftarrow (q + N), i \leftarrow 0$
43.  return
```

5.5 Conclusions

It has been demonstrated that block suffix trees are computationally less expensive representations of the suffix trees of strings whose symbols are blocks of uniform length consisting of elementary symbols. For example, binary block-8 suffix trees are efficient representations of suffix trees of strings of bytes. An efficient algorithm for constructing block suffix trees on small alphabets and arbitrary block size has been developed. In the special case of suffix trees on small alphabets when the block size equals 1, the proposed algorithm is still advantageous because of the implicit representation of the end-of-string symbol.

In the introduction to their 2003 paper [21], Kärkkäinen and Sanders claim that their algorithm is simpler than any suffix tree construction algorithm and illustrate it by a 50 line C++ implementation. Our algorithm, when implemented on bits as elementary symbols, is of comparable simplicity: Algorithm 5.4.1 is presented at a very low level of abstraction and can be translated line-by-line into a C or C++ program. In an efficient implementation for a fixed block length, one can replace the loop at line 7 by a few operations on blocks. In this case, the performance of this algorithm matches that in the appendix of [21]. The memory footprint, however, is about four times larger, which is a significant disadvantage compared to [21] for small block sizes. For large block sizes it is less significant, while the *lexicographic naming* step becomes increasingly unwieldy, requiring an advanced data structure of its own. Our algorithm, by contrast, handles large block sizes seamlessly without modification.

It is also interesting to compare our algorithm to that by Farach [16]. There are two approaches to building a suffix tree on blocks with Farach's algorithm. If the block size is a power of two (as is often the case), Farach's algorithm can be used directly to construct the corresponding *block suffix tree*, by ending the recursion at the corresponding level. For general block sizes, one can first number the blocks in lexicographic order and then use Farach's algorithm on these numbers. In this second case, the sorting of blocks becomes the computational bottleneck. As illustrated by the proof of Theorem 5.4.1, our algorithm leverages the suffix tree's structure to achieve the lexicographic ordering of blocks implicitly. Our algorithm is simpler and faster in both cases, though Farach's algorithm may also offer similar opportunities for simplification for the binary case and block sizes of powers of two. While not as general as our solution, this might actually be worth pursuing, as Farach's algorithm does not need suffix links, thus saving one pointer (out of the average four) for each explicit node, including the leaf nodes.

Chapter 6

Conclusions

In this chapter, we summarize and interpret our results and show some possible directions for future work. In particular, we provide interpretations of the results from Chapter 4 and Chapter 5 in terms of design criteria for lossless compression applications, and revisit the experimental results in Chapter 3 to provide a better explanation for our findings in the light of the results in Chapter 4.

6.1 Summary and Interpretation of our Results

The main question that our results can help answering is whether or not it is worth doing lossless compression on the level of elementary symbols.

One one hand, we have proved an upper bound on the redundancy resulting from not taking blocks into account and also proved that strongly universal source codes for higher order Markov sources are universal for finite-memory block Markov sources as well. Thus, irrespective of the actual block size of the source, we have guaranteed performance if we use such codes.

Theorem 4.2.1 gives the best mth order Markovian approximation for a block Markov source and an explicit expression for the divergence rate resulting from this approximation. Corollary 4.2.1 shows that it converges to zero, as m, the memory of the model approaches infinity. Theorem 4.3.1 shows that this convergence rate is exponentially fast. Finally, in Theorem 4.4.1 an upper bound on the actual coding redundancy is shown.

On the other hand, when using codes that match the known block size of the source, we may have faster convergence. This is well illustrated by the experiments in Chapter 3 and the result of the follow-up experiment depicted on Figure 4.4.

In this follow-up experiment, we used the same modeling and encoding algorithm for all block sizes for a fair comparison. It is interesting to note that `bzip2`'s sophisticated modeling and encoding algorithm for the binary case performs slightly worse than the simple 0-order run-length encoding used for the experiments in Chapter 3. This can be explained by the fact that some of the output it produces is redundant for the binary alphabet.

If the block size is known, it is advantageous to take it into account (see Remark (ii) to Theorem 4.4.1). No improved modeling and encoding algorithm on the binary BWT output could have matched the performance of a similar block-sorting compression algorithm, when its block size assumption was met.

As discussed in [14], the main advantage of block-sorting compression when compared to other methods with similar performance is the low computational complexity. We have shown in Chapter 5 that block sorting remains computationally inexpensive when done on blocks of symbols rather than individual symbols. The results are applicable beyond block-sorting as well. For example, the proposed data structure and the algorithm for its construction can be applied with minimal modifications to the compression algorithm presented in [13]. Instead of a block suffix tree, a block prefix tree will need to get built in a similar fashion, generalizing the algorithm for arbitrary block sizes while retaining its complexity and performance characteristics.

Theorem 5.4.1 implies that the proposed data structure, the block-N suffix tree, is an equivalent representation of the suffix tree on the N long blocks of the input.

For the binary case, we provide a particularly efficient representation.

By carefully exploiting the advantages resulting from operating on a binary alphabet and designing a very simple and efficient suffix tree construction algorithm (Algorithm 5.4.1), we have successfully challenged the claim in [21] that the *skew* algorithm presented in that paper is simpler than any linear suffix tree construction algorithm.

6.2 Possible Directions for Future Work

Many questions regarding redundancy resulting from mismatched block size in the source and the approximating model remains open. As illustrated by the following example, it is not even guaranteed to exceed that by a symbol-level approximation.

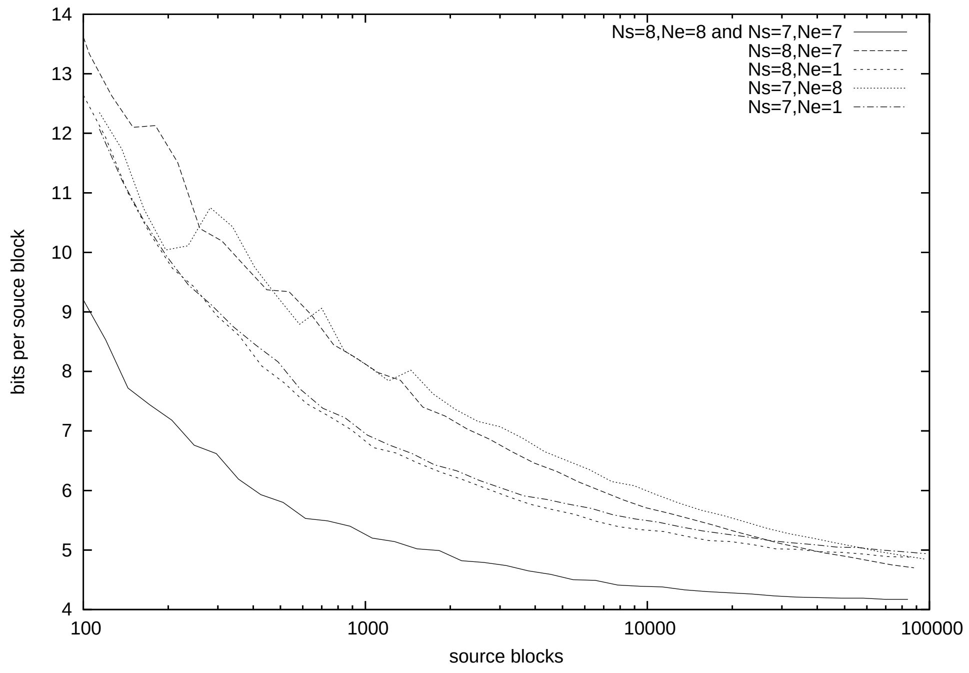

Figure 6.1: Synthetic block Markov source with first order statistics of English

The graph in Figure 6.2 shows the results of compressing a synthetic block Markov source that matches the first order statistics of `book1` from the Calgary Corpus, when representing characters as 7 and 8 bit blocks ($N_s = 7$ and N_s=8, respectively). It has been encoded using `bzip2`, operating on 1, 7 and 8 bit blocks. At around $n = 10^5$ symbols, the mismatched model surpasses the binary one. While this may be partly due to the inefficiency of `bzip2` on the binary alphabet (our compression algorithm from Chapter 3 compresses the first 10^5 blocks of this synthetic source to within 4.5 bits per block in both cases), it certainly invites further studies.

The theoretical problem underlying these experimental results is the following: Assuming a block-N Markov source and universal codes that provide near optimal redundancy rates for order m Markov sources, what is the *minimax redundancy* in this scenario? Note that the results of Chapter 4 only partially answer this question by giving a uniform upper bound for a strong family of universal codes. Missing are a lower bound and a matching upper bound (likely tighter than that of Section 4.4).

In cryptography, the results in Chapter 4 can be applied to mounting a known ciphertext attack against encrypted sources with known cleartext statistics, when the encryption is realized with a block cipher used in ECB (electronic codebook) mode, but the block size of the cipher is not known in advance.

As noted in Chapter 5, it would be interesting to investigate whether Farach's algorithm [16] offers similar simplification opportunities for the binary case. While that would limit the attainable block sizes to powers of two, it would also offer considerable savings in memory costs by omitting suffix links. It would also be interesting to study block suffix tree construction under various advanced models of computation, as it definitely offers many opportunities for parallel processing.

Bibliography

[1] A. Albers, M. Mitzenmacher: "Average case analyses of list update algorithms, with applications to data compression", *Algorithmica* vol. 21, pp. 312–329, 1998.

[2] V. Anantharam: "A large deviations approach to error exponents in source coding and hypothesis testing," *IEEE Trans. Information Theory*, vol. 36, pp. 938–943, July 1990.

[3] Z. Arnavut, S. Magliveras: "Block sorting and compression"*, Proceedings of the DCC*, pp. 181–190, 1997.

[4] J. L. Bentley, R. Sedgewick: "Fast algorithms for sorting and searching strings"*, Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 360–369, 1997.

[5] R. de la Brandais: "File searching using variable length keys." *Proceedings of Western Joint Computer Conference*, vol. 15, pp. 295–298, 1959.

[6] M. Burrows, D. J. Wheeler: "A block-sorting lossless data compression algorithm"*, SRC Research Report 124*, Digital Systems Research Center, Palo Alto, CA, May 1994.

[7] B. Chapin, S. R. Tate: "Higher compression from the Burrows-Wheeler transform by modified sorting" *http://www.cs.unt.edu/~srt/papers/bwtsort.pdf*, 1998.

[8] T. M. Cover, J. A. Thomas: “Elements of Information Theory” *Wiley Series in Telecommunications*, New York, 1991.

[9] I. Csiszár, J. Körner: “Information Theory” Academic Pres New York, 1981.

[10] I. Csiszár, T. M. Cover, B.-S. Choi: “Conditional limit theorems under Markov conditioning,” *IEEE Trans. Information Theory*, vol. 33, pp. 788–801, Nov. 1987.

[11] I. Csiszár, P. C. Shields: “Information Theory and Statistics: A Tutorial”, *Foundations and Trends in Communications and Information Theory*, vol. 1, no. 4, pp. 417-528, Dec. 2004.

[12] S. Deorowitz: “Second step algorithms in the Burrows-Wheeler compression algorithm”*, Software — Practice and Experience*; vol. 32, pp. 99–111, Feb. 2002.

[13] M. Effros: “PPM performance with BWT complexity: a fast and effective data compression algorithm” *Proceedings of the IEEE — Special Issue: Lossless Data Compression*, vol. 88, pp. 1703–1712, Nov. 2000.

[14] M. Effros, K. Visweswariah, S. R. Kulkarni, S. Verdú: “Universal lossless source coding with the Burrows-Wheeler transform” *IEEE Transactions on Information Theory* vol. 48, pp. 1061–1081, May 2002.

[15] P. Elias: “Universal codeword sets and representations of integers” *IEEE Transactions on Information Theory*, vol. 21, pp. 194–203, 1975.

[16] M. Farach: “Optimal suffix tree construction with large alphabets” *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pp. 137–143, 1997.

[17] R. Giegerich, S. Kurtz: “From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction” *Algorithmica,* vol. 19, pp. 331–353, 1997.

[18] S. W. Golomb: “Run-length encodings” *IEEE Transactions on Information Theory*, vol. 12, pp. 399–401, 1966.

[19] R. M. Gray: “Entropy and Information Theory” Springer-Verlag, New York, 1990.

[20] P. Harremoë, F. Topsøe: “Zipf’s law, hyperbolic distributions and entropy loss”, *http://www.math.ku.dk/~topsoe/ISIT2002procZipf.pdf*, 2002.

[21] J. Kärkkäinen, P. Sanders: “Simple linear work suffix array construction”, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pp. 943–955, 2003.

[22] R. E. Krichevsky: “Optimal source coding based on observation” *Problems of Information Transmission*, vol. 11, pp. 37-48, 1975. (in Russian)

[23] R. E. Krichevsky, V. K. Trofimov: “The performance of universal encoding” *IEEE Transactions on Information Theory*, vol. 27, pp. 199–207, 1981.

[24] S. Kullback, R. A. Leibler: “On information and sufficiency” *Ann. Math. Stat.*, vol. 22, pp. 79–86, 1951.

[25] G. G. Langdon, J. J. Rissanen: “A simple general binary source code” *IEEE Transactions on Information Theory*, vol. 28, p. 800, 1982.

[26] G. G. Langdon: “An introduction to arithmetic coding” *IBM Journal of Research and Development*, vol. 28, pp. 135–149, 1984.

[27] B. B. Mandelbrot: “On the theory of word frequencies and on related Markovian models of discourse”, in R. Jacobsen (ed.): *“Structures of Language and its Mathematical Aspects”*, AMS, New York, 1961.

[28] E. M. McCreight: “A space-economical suffix tree construction algorithm” *JACM*, vol. 23, pp. 262–272, Feb. 1976.

[29] N. Merhav and J. Ziv: "A Bayesian approach for classification of Markov sources", *IEEE Trans. Information Theory*, vol. 37, pp. 1067–1071, July 1991.

[30] D. A. Nagy, T. Linder: "Experimental study of a binary block-sorting compression scheme", *Proceedings of Data Compression Conference DCC'03*, IEEE Comp. Soc. Press, p. 439, Snowbird, UT, 2003.

[31] D. A. Nagy, T. Linder: "Higher-order Markov modeling of block-Markov sources," *Proceedings of 22nd Queen's Biennial Symposium on Communications*, pp. 118-120, Kingston, ON, June 2004.

[32] D. A. Nagy, A. György, T. Linder: "Convergence rates in higher order Markov modeling of block-Markov sources," *Proceedings of the 2005 Canadian Workshop on Information Theory*, pp. 111-114, Montreal, QC, June 2005.

[33] D. A. Nagy, A. György, T. Linder: "Symbol-Based modeling and coding of block-Markov sources," submitted for publication in *IEEE Transactions on Information Theory* on Sep. 2, 2005.

[34] S. Natarajan: "Large deviations, hypothesis testing, and source coding for finite Markov chains," *IEEE Trans. Information Theory*, vol. 31, pp. 360–365, May 1985.

[35] P. Orstein, P. C. Shields: "Universal almost sure data compression", *Annals of Probability*, vol. 18, pp. 441–452, 1990.

[36] Z. Rached: "Information measures for sources with memory and their application to hypothesis testing and source coding" *PhD Thesis, Dept. of Math. and Stat., Queen's University*, 2002.

[37] M. Schindler: "A fast block-sorting algorithm for lossless data compression", *Proceedings of the DCC*, p. 469, 1997.

[38] C. E. Shannon: "A mathematical theory of communication" *Bell Sys. Tech. Journal*, vol. 27, pp. 379–423, 623–656, 1948.

[39] E. Ukkonen: "On-line construction of suffix trees"*, Algorithmica*, vol. 14, pp. 249–260, 1995.

[40] M. J. Weinberger, J. J. Rissanen: "A universal finite memory source", *IEEE Trans. Information Theory*, vol. 41, pp. 643–652, May 1995.

[41] P. Weiner: "Linear pattern matching algorithms." *Proceedings of IEEE 14th Annual Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[42] I. Witten, T. Bell, "The Calgary text compression corpus," available via anonymous ftp at *ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus*

[43] J. Ziv, A. Lempel: "A universal algorithm for sequential data compression" *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, 1977.

[44] J. Ziv, A. Lempel: "Compression of individual sequences via variable-rate coding" *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, May 1978.