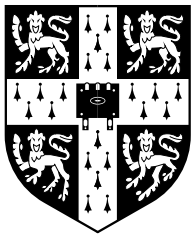
Lossless Data Compression

Christian Steinruecken
King's College



Dissertation submitted in candidature for the degree of Doctor of Philosophy,
University of Cambridge

August 2014

Typeset on May 20, 2015, using LaTeX, gnuplot, and custom-written software.

Lossless Data Compression

Christian Steinruecken

Abstract

This thesis makes several contributions to the field of data compression. Lossless data compression algorithms shorten the description of input objects, such as sequences of text, in a way that allows perfect recovery of the original object. Such algorithms exploit the fact that input objects are not uniformly distributed: by allocating shorter descriptions to more probable objects and longer descriptions to less probable objects, the expected length of the compressed output can be made shorter than the object's original description. Compression algorithms can be designed to match almost any given probability distribution over input objects.

This thesis employs probabilistic modelling, Bayesian inference, and arithmetic coding to derive compression algorithms for a variety of applications, making the underlying probability distributions explicit throughout. A general compression toolbox is described, consisting of practical algorithms for compressing data distributed by various fundamental probability distributions, and mechanisms for combining these algorithms in a principled way.

Building on the compression toolbox, new mathematical theory is introduced for compressing objects with an underlying combinatorial structure, such as permutations, combinations, and multisets. An example application is given that compresses unordered collections of strings, even if the strings in the collection are individually incompressible.

For text compression, a novel unifying construction is developed for a family of context-sensitive compression algorithms. Special cases of this family include the PPM algorithm and the Sequence Memoizer, an unbounded depth hierarchical Pitman–Yor process model. It is shown how these algorithms are related, what their probabilistic models are, and how they produce fundamentally similar results. The work concludes with experimental results, example applications, and a brief discussion on cost-sensitive compression and adversarial sequences.

Declaration

I hereby declare that my dissertation entitled "Lossless Data Compression" is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University.

I further state that no part of my dissertation has already been or is being concurrently submitted for any such degree or diploma or other qualification.

Except where explicit reference is made to the work of others, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation does not exceed sixty thousand words in length.

Date: .....

Signed: .....

Christian Steinruecken
King's College

Contents

Notation

$\emptyset$	The empty set.
$\{a, b, c\}$	The set of elements a, b and c.
$\{a\!:\!2, b\!:\!1\}$	The multiset containing two occurrences of a, and one occurrence of b. The same multiset could also be written $\{a, a, b\}$.
$\mathbb{N}$	The set of natural numbers $\{0, 1, 2, \dots\}$.
$\mathbb{R}$	The set of real numbers.
$[a, b]$	The closed real interval from a to b. Formally, r is in the set $[a, b]$ if and only if r is in the set $\mathbb{R}$ and $a \leq r \leq b$.
$[a, b)$	The semi-open interval of real numbers r satisfying $a \leq r < b$.
$(a, b]$	The semi-open interval of real numbers r satisfying $a < r \leq b$.
(a, b)	The open interval of real numbers r from a to b, satisfying $a < r < b$.
$\lfloor x \rfloor$	The largest integer smaller than or equal to x.
$\lceil x \rceil$	The smallest integer larger than or equal to x.
$\mathcal{A}$	Set or multiset variable.
$\mathcal{A}(x)$	Indicator function of set $\mathcal{A}$, or multiplicity function of multiset $\mathcal{A}$. $\mathcal{A}(x)$ denotes how often x occurs in $\mathcal{A}$. For sets, $\mathcal{A}(x) \in \{0, 1\}$. For multisets, $\mathcal{A}(x) \in \mathbb{N}$.
$x \in \mathcal{A}$	x is an element in set or multiset $\mathcal{A}$. The statements $x \in \mathcal{A}$ and $\mathcal{A}(x) \neq 0$ are equivalent.
$\lvert\mathcal{A}\rvert$	Cardinality of set (or multiset) $\mathcal{A}$. For finite $\mathcal{A}$, the cardinality $\lvert\mathcal{A}\rvert$ equals $\sum_{x \in \mathcal{A}} \mathcal{A}(x)$.
$\mathcal{A} \cup \mathcal{B}$	Union of sets $\mathcal{A}$ and $\mathcal{B}$, with the property that $(\mathcal{A} \cup \mathcal{B})(x) = \max(\mathcal{A}(x), \mathcal{B}(x))$.
$\mathcal{A} \uplus \mathcal{B}$	Multiset-union of $\mathcal{A}$ and $\mathcal{B}$, with the property that $(\mathcal{A} \uplus \mathcal{B})(x) = \mathcal{A}(x) + \mathcal{B}(x)$. Also, $\lvert\mathcal{A} \uplus \mathcal{B}\rvert = \lvert\mathcal{A}\rvert + \lvert\mathcal{B}\rvert$.
$\mathcal{A} \cap \mathcal{B}$	Intersection of sets $\mathcal{A}$ and $\mathcal{B}$, with the property that $(\mathcal{A} \cap \mathcal{B})(x) = \min(\mathcal{A}(x), \mathcal{B}(x))$.

$\mathcal{A} \setminus \mathcal{B}$	Set containing the elements of $\mathcal{A}$ but not the elements of $\mathcal{B}$.
$\mathcal{A}\times\mathcal{B}$	Cartesian product of sets $\mathcal{A}$ and $\mathcal{B}$, with the property that $\lvert\mathcal{A}\times\mathcal{B}\rvert = \lvert\mathcal{A}\rvert \cdot \lvert\mathcal{B}\rvert$.
$\mathcal{A} \to \mathcal{B}$	Set of functions from $\mathcal{A}$ to $\mathcal{B}$. Specifically, f is in the set $(\mathcal{A} \to \mathcal{B})$ if and only if for every a in $\mathcal{A}$ there exists a unique b in $\mathcal{B}$ such that $(a, b) \in f$.
$2^{\mathcal{A}}$	Powerset of $\mathcal{A}$.
$\mathcal{A}^n$	n-dimensional vector space over $\mathcal{A}$.
$\mathcal{A}^*$	Union of all finite dimensional vector spaces over $\mathcal{A}$, i.e. $\mathcal{A}^* = \bigcup_{n=1}^{\infty} \mathcal{A}^n$.
$\mathcal{A}^\infty$	Countably infinite-dimensional vector space over $\mathcal{A}$.
$x_1 \ldots x_N$	A sequence of N elements, shorthand for $x_1,\ x_2,\ \ldots,\ x_N$.
ε	The empty sequence.
δ_x	Dirac delta function at point x, with $\int \delta_x \, dx = 1$.
$\mathbb{1}[\ Q\ ]$	Indicator function. $\mathbb{1}[\ Q\ ]$ equals 1 when predicate Q is true, and 0 otherwise.
$\#[\ Q_k\ ]_{k=1}^{K}$	Counting function, pronounced "count Q_k". Equals the number of times a predicate Q_k is true for given boundary conditions. $\#[Q_k]_{k=1}^{K}$ equals $\sum_{k=1}^{K} \mathbb{1}[Q_k]$.
D	A probability distribution or measure.
$D(x)$	The probability mass of a value x under distribution D.
$D(\mathcal{S})$	The total probability mass of a set $\mathcal{S}$ of outcomes. $D(\mathcal{S}) = \sum_{x\in\mathcal{S}} D(x)$.
$D_{\setminus\mathcal{S}}$	Probability distribution D excluding elements in set $\mathcal{S}$. For any x with support in D, $D_{\setminus\mathcal{S}}(x)$ equals 0 if $x \in \mathcal{S}$, and $D(x) \cdot (1 - D(\mathcal{S}))^{-1}$ otherwise.
D_Σ	Lower cumulative mass function of distribution D. An implicit total ordering $\sqsubset$ is assumed. For any element x with support in D, $D_\Sigma(x)$ equals $\sum_{y\sqsubset x} D(y)$.
D_Σ^+	Upper cumulative mass function of distribution D. For any element x with support in D, $D_\Sigma^+(x) = D_\Sigma(x) + D(x)$. Equivalently, $D_\Sigma^+(x) = \sum_{y\sqsubseteq x} D(y)$.

$\mathrm{KL}(P \,\|\, Q)$	Kullback–Leibler divergence between distributions P and Q (Kullback and Leibler, 1951). For discrete P and Q, $\mathrm{KL}(P \,\|\, Q) \stackrel{\text{def}}{=} \sum_{x \in \mathcal{X}} P(x) \ln \frac{P(x)}{Q(x)}$.
$\mathbb{E}_x^P[\, f_x \,]$	The expected value of a quantity f_x that depends on a P-distributed variable x. For discrete P and $x \in \mathcal{X}$, $\mathbb{E}_x^P[f_x] \stackrel{\text{def}}{=} \sum_{x \in \mathcal{X}} P(x) \cdot f_x$.
$h_P(x)$	The Shannon information content of a value x under probability distribution P, defined as $h_P(x) \stackrel{\text{def}}{=} \log_2 \frac{1}{P(x)}$.
$\mathrm{H}(P)$	The Shannon information entropy of a probability distribution P, defined as $\mathrm{H}(P) \stackrel{\text{def}}{=} \mathbb{E}_x^P[h_P(x)]$.
$X \sim D$	Random variable X, distributed according to distribution D.
$\Pr(X)$	The probability distribution of random variable X. If $X \sim D$, then $\Pr(X) = D$.
$\Gamma(x)$	Value of the Gamma function at point $x \in \mathbb{R}$. The Gamma function is defined as $\Gamma(x) = \int_0^\infty w^{x-1} e^w \, dw$, and satisfies $\Gamma(x+1) = x \cdot \Gamma(x)$.
C	A code C that maps elements of a set $\mathcal{X}$ to finite sequences of some finite alphabet $\mathcal{A}$. C is a function in $\mathcal{X} \to \mathcal{A}^*$. $C(x)$ denotes the code word assigned to element x, and $\lvert C(x) \rvert$ its code word length.
iid	Text abbreviation meaning "independent and identically distributed".
`EOF`	End-of-file symbol.

Bayes' theorem

$$\Pr(X \mid Y) = \frac{\Pr(Y \mid X) \cdot \Pr(X)}{\Pr(Y)}$$

Meaning of syntax colours

- Probability distributions
- Sets and multisets
- Codes
- Sequences and vectors

Chapter 1

Introduction

1.1 Codes and communication

The communication of large amounts of information has become a defining feature of our civilisation. In our modern age, information is increasingly represented in a digital form, and its transmission happens mainly electronically. With the rise of the Internet, the ability to exchange data globally has come to an ever increasing proportion of the world population.

There are many ways of representing information digitally, and any particular choice of representation is called a code. A code defines a mapping between a set of input objects $\mathcal{X}$ and the set of sequences of symbols from a finite alphabet $\mathcal{S}$. Codes can be designed to serve particular purposes, for example:

- Standardising a way of communicating. Notable examples include the ASCII code (ASA, 1963), or the Unicode standard (Unicode, 1991).

- Encoding an object x in such a way that its output sequence s is robust to transmission errors, with the goal that x can be reconstructed from s even when some of the output sequence has been altered during transmission. These are *error correcting codes*.

- Producing a compact representation of an object x, minimising the length of the output sequence. This is called *data compression*, and requires knowledge about the probability distribution over input objects. The better the compressor's assumed distribution fits the actual distribution, the better the input objects can be compressed.

- Securing an object x against interception. This is called *encryption*, and it works by making the decoding procedure depend on a secret without which decoding would be impossible or computationally infeasible.

Codes can be chained together such that the output of one code is the input to another code. For example, an object x might first be compressed, then encrypted and finally secured against transmission errors, combining three separate codes. Success stories where such combined codes are used include communication with distant man-made objects, such as space satellites or robots on other planets of our solar system.

Coding theory originated with the work of Claude Shannon (1948), which gives a formal way of measuring information and uncertainty. Shannon's work laid the foundations for error correcting codes and data compression, and for encryption one year later. Among his results is a proof that for any channel, there are codes that can achieve near error-free communication over that channel. The availability of error-free communication is assumed throughout this thesis.

1.2 Data compression

A data compression algorithm translates an input object to a compressed sequence of output symbols, from which the original input can be recovered with a matching decompression algorithm. Compressors (and their matching decompressors) are designed with the goal that the compressed output is, on average, cheaper to store or transmit than the original input.

Most contemporary systems transmit information using a binary alphabet, such as $\{0, 1\}$. When the transmission costs of a `0` or a `1` are equal, the task of data compression is identical to the task of minimising the message length. (Compressing to output alphabets with unequal transmission costs is discussed in chapter 8; everywhere else in this thesis, equal transmission costs are assumed.)

Data compression algorithms exploit that the occurrence probability of equally sized input objects is not uniform: by allocating shorter descriptions to probable objects and longer descriptions to less probable objects, transmission cost can be saved on average. At least implicitly, any compression algorithm necessarily makes assumptions about which inputs are more probable than others. These assumptions can be expressed in the form of a probability distribution over input objects, which is called the algorithm's *implicit probability distribution*. (Details are given in chapter 3.)

A lossless compression code is *optimal* for a given probability distribution if the expected output length for a random input message is minimal. The theoretical limit of that minimum is equal to the *Shannon information entropy* of the distribution over messages. There are algorithms that can create near-optimal compression codes for nearly any given probability distribution, getting within 2 bits of the theoretical limit. Examples of such methods include the Huffman algorithm (covered in section 2.1.1) and the family of arithmetic coding

algorithms (covered in chapter 3).

This thesis advocates constructing compression algorithms directly from probabilistic models, using arithmetic coding for the code generation. With a probabilistic model, all assumptions and uncertainty about the input data are made explicit, making it easier to reason about the properties of the resulting compressor. Furthermore, the law of inverse probability (Bayes' theorem) allows probabilistic models to be combined in a principled and consistent way, enabling the modular construction of compression algorithms from smaller components.

Lossless versus lossy compression

An important distinction is made between *lossy* and *lossless* compression methods:

1. **Lossy compression**: preserves the important part of the original message, and discards unimportant details. The foundations of lossy compression were laid in Shannon's rate–distortion theory (Shannon, 1948). Well-known examples of lossy compression include perceptual coding such as the JPEG picture format (JPEG, 1992), which approximates a digital photograph using a quantized discrete cosine transform; or the MP3 audio format (MPEG, 1991; Brandenburg, 1999), which exploits psychoacoustic properties of human hearing to reduce bandwidth for inaudible frequency bands. For data where an approximate reconstruction is good enough, lossy compressors can achieve unrivalled results. Lossy compression was motivated by e.g. Blasbalg and van Blerkom (1962) and initially called "entropy reduction".

2. **Lossless compression** preserves the original message in its entirety, allowing it to be recovered exactly from the compressed output. This thesis focuses on lossless compression, although some of the structural compression methods introduced in later chapters may have interpretations as lossy compressors.

Note that a lossless data compressor cannot make every possible input string shorter – if such an algorithm existed, its recursive application could reduce every input to length zero, violating the constraint that the transformation be lossless. Every lossless compression algorithm must therefore occasionally produce output sequences that are longer than the input sequence.[1]

[1] If worst-case behaviour is a concern, one can limit by how many bits the output sequence may be longer than the input sequence. Such a modification compromises compactness of the resulting code, and therefore comes at the cost of making all output sequences longer.

Example. Consider *any* algorithm that maps input sequences to output sequences. Modify this algorithm as follows: Whenever the output sequence is longer than the input sequence, send a single `0` followed by the bit string of the original input sequence, otherwise send a `1` followed by the compressed output sequence. The total length of the final output never exceeds the length of the input by more than one bit.

Modifications of this form are used in e.g. the DEFLATE algorithm by Katz and Burg (1993).

Examples of the sort of ideas used in constructing lossless data compression algorithms can be found on page 19.

Design philosophy of compression algorithms

It is worth noting that there are different philosophical approaches to compression. One approach is to design *any* algorithm that losslessly maps input to output data, and then prove that the produced output has desirable properties. Commonly investigated properties include worst-case behaviours of the algorithm, resource costs, and the asymptotic limit of the compression rate. Notable examples of algorithms in this category include the LZ77 and LZW methods, and compressors using the Burrows–Wheeler transform (explained in chapter 2). Such algorithms do not necessarily represent (or compute) the probability distribution over input objects explicitly.

A second approach is to model the input data directly with a probability distribution, and construct an algorithm that uses this distribution (or an approximation thereof) to compute the compressed output data. Methods of this kind are typically easier to combine, modify and reason about, as the coding is cleanly separated from the modelling. Notable examples include algorithms from the PPM family (explained in chapter 6), the CTW family,[2] and the PAQ family (not covered in this thesis).[3]

These two philosophies are not necessarily disjoint. The approach taken in this thesis is the one of explicit probabilistic modelling and arithmetic coding.

1.3 Outline of this thesis

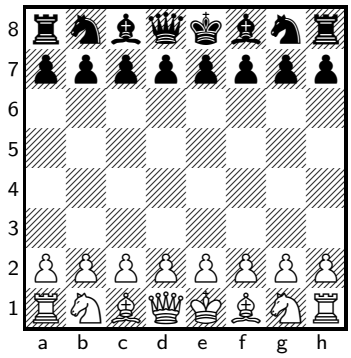
The remainder of this thesis is organised as follows. Chapter 2 introduces basic concepts and reviews selected classic compression algorithms. Chapter 3 presents arithmetic coding, the fundamental coding algorithm that underlies most of the compression techniques discussed in later chapters. For concreteness, the implementation of the arithmetic coder used in this thesis is included as JAVA source code. Chapter 4 describes several basic adaptive compression methods that use Bayesian inference for learning an unknown symbol distribution directly from the input data. Proof is given that a sequence of independent and identically distributed symbols is equally well compressed by an adaptive technique and a batch technique that compresses the sequence's symbol histogram and permutation separately.

[2]The CTW algorithm was developed by Willems et al. (1993, 1995); Willems (1998). For details of CTW and comparisons with PPM, see e.g. Åberg and Shtarkov (1997); Åberg et al. (1998); Sadakane et al. (2000); Begleiter et al. (2004); Begleiter and El-Yaniv (2006).

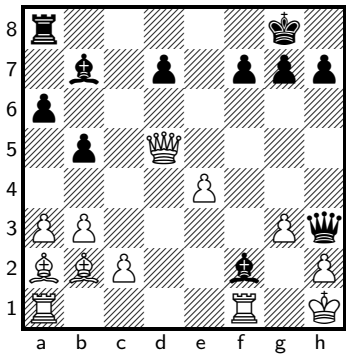
[3]Compressors in the PAQ family combine predictions from many probabilistic models (including PPM-like models) to form a single prediction. The PAQ family of compressors was developed by Mahoney (2000, 2002, 2005).

A compression puzzle

Consider the task of communicating the state of a chess board using the fewest possible number of bits. A chess board has 8×8 squares, and each square may either be empty, or contain exactly one chess piece. Each chess piece is one of six different figures {♔, ♕, ♖, ♗, ♘, ♙} and has one of two colours {■, □}. At the start of a game of chess, there are 32 pieces on the board, arranged as shown in Figure 1.1 below.



(1.1) Opening configuration of a game of chess.



(1.2) Configuration of a game in progress.

The reader is encouraged to take a moment to think of possible solutions.

Idea 1. There are 64 squares, and each can have one of 13 possible states: empty, or one of the six black or six white chess pieces. One way of describing the board would be to encode, in some fixed order, the state of each square. If 4 bits are used to encode one of the 13 possible states, the total number of bits for any board configuration would be $64 \times 4 = 256$.

Idea 2. Empty is the most frequent state, it occurs usually at least 32 times (out of 64 possible squares). So for every square (in some fixed order), we could store a single bit 0 or 1 indicating if the square is empty. If the square is non-empty, store which of the 12 pieces it contains (using 4 bits). For a chess board which has the maximum of 32 pieces, that makes $64 + 32 \times 4$ bits $= 192$ bits in total. Boards with fewer pieces are cheaper, and an empty chessboard costs 64 bits.

Idea 3. Generalising idea 2 leads to a Huffman code over the 13 possible states of each square. The Huffman algorithm is described in section 2.1.1. Even more efficient coding schemes are possible with arithmetic coding.

Idea 4. A state of a chess game can never legally have more than two ♔, or more than 16 ♙. An algorithm that excludes illegal board configurations can encode legal states more compactly. One such technique is exclusion coding (described in section 3.5.2).

Idea 5. Not all board configurations are equally probable. Assuming a probability distribution over board configurations, one could compress a random board configuration with a suitable derived arithmetic code.

Chapter 5 introduces and motivates structural compression methods; derives concrete solutions for a selection of fundamental combinatorial objects; and shows how these can be combined.

Chapter 6 reviews context-sensitive compression methods, generalises the PPM algorithm and shows its relation to the unbounded-depth hierarchical Pitman–Yor sequence model known as Sequence Memoizer. Quantitative performances of these methods are compared for a number of corpora.

Chapter 7 describes compression algorithms for multisets of sequences, a concrete application of the structural compression techniques developed in chapter 5. Finally, chapter 8 concludes with a brief treatment of cost-sensitive compression and the construction of adversarial sequences.

Chapter 2

Classical compression algorithms

This chapter reviews some well-known classical compression algorithms, highlighting some of the fundamental compression problems encountered in practice. These algorithms have set foundational trends for the majority of modern compression techniques, and many of them are still used in practice today. Discussing these methods introduces vital concepts used throughout this thesis, and motivates the more modern approaches discussed later.

The algorithms were carefully selected to be easy to understand, and are representative members of the family of bit-aligned coding methods. Despite their algorithmic simplicity, the underlying assumptions and probabilistic interpretation of these compressors may not be immediately obvious. Where possible, the probabilistic properties are pointed out.

Many classical compression methods were the result of experimentation and carefully applied heuristics. Most compressors in this chapter do not cleanly separate the tasks of modelling and coding. By contrast, modern approaches to compression often involve the use of a probabilistic model and an arithmetic coding method. These modern approaches are discussed in chapter 3, and are in many cases natural generalisations of concepts introduced here.

Some of the algorithms described here are used for baseline comparison throughout the rest of this thesis.

Notation primer

For the techniques that follow, every input file will be treated as a finite sequence of input symbols $x_1 \dots x_N$. The set of input symbols $\mathcal{X}$ is called the *source alphabet*, and is often taken to be the set of bytes $\{00_{16}, \dots FF_{16}\}$. The set of output symbols $\mathcal{Y}$ is called the *target alphabet* and is usually the set of bits $\{0, 1\}$. In practice, the sequence of output bits has to be padded to the nearest multiple of 8, so it can be stored as a file, but we'll gloss over these practical concerns for now and just focus on the essence of the algorithms.

2.1 Symbol codes

A *symbol code* C maps each possible input symbol $x \in \mathcal{X}$ to a fixed length code word $C(x) \in \mathcal{Y}^*$ of output symbols. This mapping must have the property that any concatenated sequence of code words is uniquely decodable to the corresponding sequence of input symbols. A symbol code can then be used to encode a sequence $x_1 \ldots x_N$ by concatenating the code words $C(x_n)$ of each symbol in the sequence. Code words may vary in length, which makes it possible to represent frequently occurring input symbols with shorter code words, and less frequently occurring input symbols with longer code words.

It is generally desirable for the code words to be chosen in a way that makes the decoding process simple and efficient. This can be done by enforcing a *prefix property* on the code words, which requires that no code word may be a prefix of another. In a symbol code with the prefix property, each symbol x can be decoded immediately once the next code word $C(x)$ is recognised in the input stream, as no other code word can have $C(x)$ as a prefix. The prefix property makes the termination of each code word easy to detect.

All symbol codes discussed here have the prefix property, and may also be called *prefix codes*. Non-prefix codes have no advantage over prefix codes, and for every uniquely decodable non-prefix code there exists an equivalent prefix code with matching code word lengths. An encoding and decoding procedure for prefix codes is shown in code listing 2.1.

Symbol Coding

ENCODING	**DECODING**
While there are more input symbols, repeat:	Initialise $w \leftarrow ()$. While not `EOF`, repeat:
1. Read input symbol x.	1. Read the next symbol y.
2. Find code word $C(x)$.	2. Update $w \leftarrow w :: y$.
3. Output $C(x)$.	3. If $w = C(x)$ for some symbol x, set $w \leftarrow ()$ and output x.
	If $w \neq ()$, report error "final buffer not empty".

Code listing 2.1: A general encoding and decoding procedure for symbol codes with the prefix property.

As noted in section 1.2, any code implicitly defines a probability distribution for which it is optimal. Because a symbol code maps each input symbol x to a code word comprising an integer number of output symbols, the implicit probability mass of each symbol is an integer power of $|\mathcal{Y}|^{-1}$, where $|\mathcal{Y}|$ is the size of the output alphabet. If the output alphabet $\mathcal{Y}$ is

the binary alphabet $\{0, 1\}$, a symbol code's implicit distribution D over source symbols is a *dyadic* distribution. A dyadic distribution is one whose probability masses are inverse powers of two:

$$\forall x \; \exists k \quad D(x) = \frac{1}{2^k} \tag{2.1}$$

For each dyadic distribution D there exists an optimal (though not necessarily unique) binary symbol code with the property that the length of each code word $C(x)$ is exactly equal to the symbol's Shannon information content $h_D(x)$. Implicit probability distributions and Shannon information will be defined in section 3.1.

2.1.1 Huffman coding

More generally, given an arbitrary probability distribution $P(x)$ over a *finite* set of input symbols, one can approximate P with the best matching dyadic distribution D, using an algorithm published by Huffman (1952). The main idea is to give short code words to common symbols; if symbol `B` occurs as often as symbols `A` and `C` put together, `B`'s code word should be shorter than those of `A` and `C`. The Huffman algorithm constructs an optimal binary prefix code for P. The Huffman algorithm is explained in code listing 2.2.

For any symbol distribution P, the Huffman algorithm constructs a code with the shortest possible expected code word length for a *single* symbol. The resulting code's implicit probabilities are the best possible approximation to P within the restrictive confines of binary symbol coding. But symbol codes do not optimally encode *sequences* of symbols, unless the symbol probabilities are integer powers of $|\mathcal{Y}|^{-1}$; this is a consequence of the practice that code words are simply concatenated, and of the limitation that the code word lengths are integers.

The restrictions of symbol codes are lifted by *arithmetic codes*, which assign an output sequence to the entire input message rather than to individual input symbols. Arithmetic coding is described in section 3.2. But symbol codes are still widely used in practice, due to their simplicity and ease of implementation, as well as computational speed. The Huffman algorithm is the method of choice for constructing a fixed symbol code.[1]

Adaptive variants of Huffman codes exist (Gallager, 1978; Knuth, 1985; Vitter, 1987), where instead of precomputing a fixed Huffman tree for a known distribution, both sender and receiver use a tree with weighted nodes that changes dynamically with the data. The use of adaptive Huffman codes is not recommended, as its performance is outclassed by arithmetic coding in almost every way.

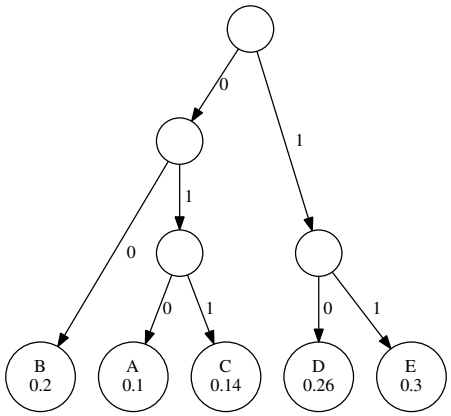
[1] For a particularly beautiful and efficient implementation, see Moffat and Katajainen (1995).

The Huffman Algorithm

Input: $\mathcal{X}$ and P. **Output:** a binary tree. **Working space:** a set $\mathcal{S}$ of trees and weights.

1. Create an empty set $\mathcal{S}$ of tuples (t, θ), where t is a (node in a) binary tree, and θ is an associated weight.
2. For each $x \in \mathcal{X}$, create a leaf node t_x and weight $\theta_x = P(x)$, and insert (t_x, θ_x) into $\mathcal{S}$. The weights sum to 1.
3. While $|\mathcal{S}| > 1$, repeat:
   - Find the two tuples (t_A, θ_A) and (t_B, θ_B) with the smallest weights in $\mathcal{S}$. (If several tuples qualify, choose deterministically.)
   - Remove (t_A, θ_A) and (t_B, θ_B) from $\mathcal{S}$.
   - Create a new tree node t_C whose children are t_A and t_B.
   - Insert $(t_C,\ \theta_A + \theta_B)$ into $\mathcal{S}$.
4. The single element left in $\mathcal{S}$ is a binary tree corresponding to an optimal prefix code.

Example. A Huffman tree generated from a 5-letter alphabet with associated probabilities.



Symbol	Prob	Implied	Code word
A	0.1	0.125	010
B	0.2	0.25	00
C	0.14	0.125	011
D	0.26	0.25	10
E	0.3	0.25	11

Code listing 2.2: The Huffman algorithm (Huffman, 1952) for generating an optimal prefix code. Given a set $\mathcal{X}$ of symbols and a probability mass function P, the algorithm constructs a binary tree representing an optimal prefix code. The binary code word of any symbol can be found by tracing the path from the root node to the symbol node, encoding each binary choice using a single bit digit $\in \{0, 1\}$.

2.2 Integer codes

This section describes some notable classical methods for encoding integers, i.e. elements from a countably infinite set such as $\mathbb{N}$ or $\mathbb{Z}$. Integer codes feature as components in a wide range of classical compression algorithms, including dictionary-based file compressors and video codecs. The methods described here were chosen to maximise clarity of explanation, or because of prevalence in industrial applications.

An integer code C maps any input element n to a code word $C(n) \in \mathcal{Y}^*$, where $\mathcal{Y}$ is some finite alphabet, typically $\{0, 1\}$. Unlike symbols codes, integer codes must be able to deal with an infinite number of possible input symbols. The length of any code word is finite, but unbounded; typical integer codes assign longer code words to larger integers.

Integer codes should have the following properties:

- The mapping between integers and code words of integer codes should be *unique*, such that each integer maps to exactly one code word, and each code word maps to a unique integer.
- Code words should be *self-delimiting*, i.e. it should never be required to know the length of a code word in order to decode it: the symbols themselves must carry information about where any given code word ends.[2]
- The space of generated sequences should be *compact*, such that there exists a unique decoding for every possible infinite sequence of input symbols.

If all three of these properties are fulfilled, any infinite sequence of symbols $(y_1, y_2...)$ can be uniquely and efficiently translated to an infinite sequence of integers, and vice versa.

Again, for computational convenience, it helps if the code words satisfy the prefix property.

The code word lengths of an integer code define an implicit probability distribution over integers, where each integer n has probability mass that is inversely proportional to its code word length:

$$\Pr(n) = \frac{1}{2^{|C(n)|}} \tag{2.2}$$

2.2.1 Unary code

The simplest integer code is probably the *unary code*, which maps each positive integer $n \in \mathbb{N}$ to a sequence of n zeros, followed by a one:

Integer	Code word	Implied probability
0	`1`	2^{-1}
1	`01`	2^{-2}
2	`001`	2^{-3}
3	`0001`	2^{-4}
n	$\underbrace{\texttt{00...0}}_{n}\texttt{1}$	2^{-n-1}

It is easy to verify that this code satisfies the criteria from above: it has a unique mapping, its code words are self-delimiting, and it generates the entire space of possible binary sequences.

One can interpret the unary code as a series of binary questions "Is it k?" starting from $k = 0$ and incrementing k after each question. When the answer to a question is no, a `0` is written; when the answer is yes, a `1` is written and the process terminates.

[2]The requirement that code words be self-delimiting may seem a bit subtle at first. As an example, consider mapping each positive integer n to its representation in binary. Although the assignment between numbers and code words is unique, this code is not uniquely decodable, as there is no way to tell from the stream of binary digits where any given code word ends. Of course, if the lengths of the code words are known in advance (like in ASCII) or transmitted separately, then this scheme would work.

The unary code uses $n+1$ bits to encode any integer n. This code is optimal for the distribution that assigns to each integer n the probability mass of 2^{-n-1}, i.e. a geometric distribution with success parameter $\theta = \frac{1}{2}$.

2.2.2 Elias gamma code

The perhaps simplest non-unary integer code is the γ-code by Elias (1975). This code assigns code words to natural numbers greater than zero, $n \in \{1, 2, 3, \dots\}$.

The code word for integer n is formed by writing n in binary notation (without the leading `1`), prefixed by its string length written in unary, using the unary code from the previous section. This solves the termination problem, as the unary number can be decoded first – this way, the decoder knows how many binary digits to read.

For small integers, the Elias γ-code generates the following code words:

Integer	Code word	Implied probability
1	`1`	2^{-1}
2	`010`	2^{-3}
3	`011`	2^{-3}
4	`00100`	2^{-5}
5	`00101`	2^{-5}
6	`00110`	2^{-5}
7	`00111`	2^{-5}
8	`0001000`	2^{-7}
9	`0001001`	2^{-7}
n	$\underbrace{\texttt{0}\dots\texttt{0}}_{\lfloor \log_2 n \rfloor}\underbrace{\texttt{1}\dots}_{n}$	$2^{-2\lfloor \log_2 n \rfloor - 1}$

This method encodes each integer using $2\lfloor \log_2 n \rfloor + 1$ bits. To encode zero, the code can be shifted down by 1. To encode all integers $i \in \mathbb{Z}$, a suitable bijection can be used; for example, $i = \left\lfloor \frac{n}{2} \right\rfloor \cdot (-1)^{n \bmod 2}$ maps natural numbers $(1, 2, 3, 4, 5, ...)$ onto integers $(0, 1, -1, 2, -3, ...)$.

The basic idea behind this coding technique is to augment the natural binary representation of an integer with a length indicator, making the resulting code words uniquely decodable. The same basic construction is used by many other integer codes, some of which are reviewed in the remainder of this section.

2.2.3 Exponential Golomb codes

As an example of integer codes which are widely used in practice, I will describe the family of *exponential Golomb codes*. This family is really just one code, parametrised by an integer k that determines the distribution of code word lengths. In particular, the kth code distributes the probability mass such that the first 2^k integers have equal probability, and a joint mass of exactly $\frac{1}{2}$.

Table 2.1: Code words the exponential Golomb code assigns to various integers, for different settings of k. For $k = 0$, the exponential Golomb code produces the same code words as the Elias γ-code (shifted down by one integer).

Integer	Exponential-Golomb code words				Integer
(unsigned)	$k = 0$	$k = 1$	$k = 2$	$k = 3$	(signed)
0	1	10	100	1000	0
1	010	11	101	1001	1
2	011	0100	110	1010	−1
3	00100	0101	111	1011	2
4	00101	0110	01000	1100	−2
5	00110	0111	01001	1101	3
6	00111	001000	01010	1110	−3
7	0001000	001001	01011	1111	4
8	0001001	001010	01100	010000	−4
9	0001010	001011	01101	010001	5
10	0001011	001100	01110	010010	−5
11	0001100	001101	01111	010011	6
12	0001101	001110	0010000	010100	−6

An exponential Golomb code with parameter k encodes a natural number n as follows:

1. Compute $m = \left\lfloor \log_2(n + 2^k) \right\rfloor - k$, and encode m using the unary code from section 2.2.1.
2. Write the binary representation of $n + 2^k$, omitting the leading 1. This representation uses exactly $k + m$ bits.

The number m represents the number of additional binary digits needed to encode n, having made use of k "free digits". So in total, an exponential Golomb code represents any positive integer n with exactly $\left\lfloor \log_2(n + 2^k) \right\rfloor + 1$ bits. Table 2.1 shows some of the code words generated by exponential Golomb codes with different settings of k.

For example, in the case of $k = 3$, no additional digits are needed to represent the numbers 0 to 7 in binary, because the $k = 3$ free digits provide enough space. This means that $m = 0$ for these numbers, and the unary encoding of $m = 0$ is the single digit prefix 1. Numbers 8 to 15 require an additional 4th digit, so $m = 1$, giving unary prefix 01. This coding method can be used for signed integers, too, usually by alternating the sign with the least significant digit of the code word, as shown in Table 2.1.

Industrially, exponential Golomb codes are widely used, e.g. in the H.264 video compression standard (ITU-T, 2003), and the BBC's Dirac open video codec (BBC, 2008). H.264 is used to encode videos on YouTube and BluRay discs, and is implemented by most contemporary web browsers.

Exponential Golomb codes were originally introduced by Teuhola (1978) as a modification of Golomb codes (Golomb, 1966). Golomb codes were generalised by Bottou et al. (1998) to form the Z-Coder, a fast arithmetic coder for binary inputs.

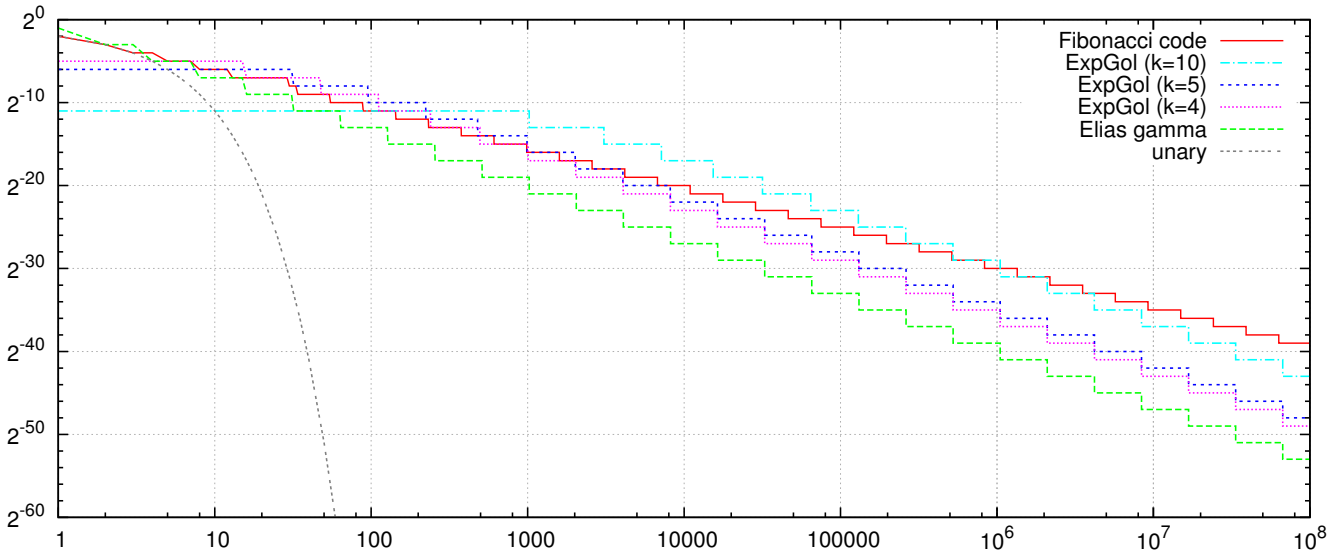


Figure 2.1: The implicit distributions over integers of various integer codes. Integers are on the x-axis, and probability mass is on the y-axis.

2.2.4 Other integer codes

Plenty of other integer codes exist. Many of these are variations of the construction shown earlier: an encoding of the integer in binary notation, prefixed by some representation of its string length, i.e. the number of binary digits. The way the string length is encoded is the main distinction among different methods.

An interesting edge case is the ω-code by Elias (1975), which encodes natural number n by prefixing its binary representation with the length m encoded (recursively) using the ω-code itself, until the base case $m = 1$ is reached; a final `0` is added to the end to mark termination. A similarly recursive code was published by Levenshtein (1968).

An example of integer codes which do not follow the above construction is the family of *Fibonacci codes* (Kautz, 1965; Apostolico and Fraenkel, 1987; Fraenkel and Klein, 1996). A Fibonacci code represents any natural number n as a sum of Fibonacci numbers, where each Fibonacci number may occur at most once.[3] The encoding procedure for n works as follows: For each Fibonacci number from $F(2)$ upwards, a `1` is written for presence or a `0` for absence of $F(k)$ in n's Fibonacci sum representation. The procedure stops after the `1`-digit for the largest contained Fibonacci number is written. The resulting code words are unique, and have the additional property that the substring `11` never occurs. Appending an additional `1` to the encoded number thus marks termination.

The implicit distributions of several integer codes are shown in Figure 2.1.

[3] See Zeckendorf (1972, Théorèmes Ia & Ib) for a proof that this representation is unique and exists for all integers.

1	11	6	10011	11	001011	16	0010011	21	00000011
2	011	7	01011	12	101011	17	1010011	22	10000011
3	0011	8	000011	13	0000011	18	0001011	23	01000011
4	1011	9	100011	14	1000011	19	1001011	24	00100011
5	00011	10	010011	15	0100011	20	0101011	25	10100011

Table 2.2: Fibonacci code words assigned to integers $n = 1 \ldots 25$ by a Fibonacci code. All code words terminate with 11. For example, $17 = F(2) + F(4) + F(7)$: $1_2 0_3 1_4 0_5 0_6 1_7 1_T$.

2.3 Dictionary coding

Some of the most successful early file compression algorithms belong to the class of *dictionary coding* methods. Dictionary coders compress their input sequence by recognising substrings that occurred earlier in the input, and coding a pointer to the earlier occurrence, rather than the repeated sequence itself. The encoder stores these substrings in memory in order of occurrence – this gradually growing list of strings is called the *dictionary* of the algorithm. The process is designed such that the dictionary constructed by the encoder can be reconstructed exactly by the decoder, along with the original symbol sequence.

2.3.1 LZW

A representative and elegant example from the class of dictionary coders is the LZW algorithm by Welch (1984). Essentially, LZW translates a sequence of characters to a sequence of pointers into a dynamically constructed dictionary. The pointers into the dictionary are positive integers and can be encoded with an integer code. A special symbol (EOF) can be added to the input alphabet for signalling the end of the sequence.

Concretely, the algorithm maintains a dictionary Φ and a counter K of the number of entries in the dictionary. Initially, Φ contains one entry for each symbol in the input alphabet $\mathcal{X}$, and K equals $|\mathcal{X}|$. Whenever a word w is added to Φ, counter K is incremented and the new word is stored at position $K-1$. For any given word w, $\Phi[w]$ returns the position at which w is stored in Φ, and similarly, for any positive integer $k < K$, $\Phi^{-1}[k]$ returns the word stored at position k.

The *compressor* maintains a symbol buffer w, which is initially equal to the first symbol of the input sequence (unless the input sequence is empty, in which case the algorithm terminates). While w is contained in the dictionary Φ, the next input symbol x is read and appended to w, until $w :: x$ is not contained in Φ (but w is). The algorithm then encodes $\Phi[w]$ using e.g. an integer code. Then, $w :: x$ is stored in Φ at position K, and K is incremented by one. Finally, buffer w is set to x (the last seen symbol), and the cycle repeats.

It is not immediately obvious that this process is reversible. In particular, the compressor

never explicitly writes x, the final symbol of the newly added word $(w :: x)$ to the encoded stream. It turns out that x can be deduced because it is also the start symbol of the next word, allowing the decompressor to complete the last dictionary entry in retrospect. Here's how this works:

The *decompressor* uses the integer code to decode the next word index k. If the total number of dictionary entries K is greater than $|\mathcal{X}|$, the last added word $\Phi^{-1}[K-1]$ in the dictionary is not yet complete, as its final symbol was unavailable in the last iteration. Since its unknown final symbol must equal the first symbol of the current word, the decoder can complete $\Phi^{-1}[K-1]$ by appending the first symbol of $\Phi^{-1}[k]$. After this operation, the dictionary is in a safe state and the decoder can look up the current word $w = \Phi^{-1}[k]$ and copy it to the output. To replicate the actions of the encoder, the next word of form $w :: y$ must be added to the dictionary, but its final letter y cannot yet be determined. So instead, the decoder adds w to Φ at position K, leaving it to be completed in the next iteration of the algorithm. Finally, K is incremented and the cycle repeats. The LZW algorithm is summarised in code listing 2.3.

2.3.2 History and significance

The family of dictionary coding algorithms started with an algorithm named LZ77 (Ziv and Lempel, 1977) and its modification LZ78 (Ziv and Lempel, 1978), published one year later. Many variants followed. Notable ones include:

- LZSS (Storer and Szymanski, 1982), which modifies LZ77 by restricting string substitutions to those which "pay off"; LZSS was used in some of the classic archiving software of the 1980s, including ARJ, RAR, and the Game Boy Advance.
- LZW (Welch, 1984), which is based on LZ78. It was used in Unix `compress` (Thomas et al., 1985), and file standards such as GIF, TIFF and PDF. Many variants of LZW exist.
- DEFLATE, which combines LZ77 with Huffman coding, was created by Katz and Burg (1993) for the `pkzip` compression utility, and described as part of the ZIP file format specification by Katz (1993), and in RFC 1951 by Deutsch (1996). DEFLATE is widely used e.g. in `gzip` / zlib (Gailly and Adler, 1992), SSH, the Linux kernel, and file formats such as PDF and PNG.
- LZMA (Pavlov, 2011), which is based on LZ77 but includes various additions, e.g. an adaptive code for literals (using arithmetic coding), and some basic mechanisms of context handling. LZMA is the primary algorithm used in `7-zip` (Pavlov, 2003), and was later incorporated into the ZIP file format (Peterson et al., 2006).

The LZW algorithm

ENCODING

A. Initialise dictionary Φ with one entry for each symbol $x \in \mathcal{X}$, set $K \leftarrow |\mathcal{X}|$, and $w \leftarrow \varepsilon$ (the empty sequence).

B. While there are more input symbols:

1. Read input symbols until we've found the longest sequence w still contained in Φ, so that $(w :: x)$ is not in Φ.
2. Find index $k = \Phi[w]$, and encode it.
3. Add $(w :: x)$ to Φ, at index K.
4. $K \leftarrow K + 1$.
5. $w \leftarrow x$. (Note that x has not yet been transmitted.)

C. If w is non-empty, encode $\Phi[w]$.

DECODING

A. Initialise dictionary Φ with one entry for each symbol $x \in \mathcal{X}$, set $K \leftarrow |\mathcal{X}|$, and $w \leftarrow \varepsilon$ (the empty sequence).

B. While there is more to decode, repeat:

1. Decode index k.
2. Look up the word w from $\Phi^{-1}[k]$.
3. If $K > |\mathcal{X}|$, then complete word $K - 1$ in Φ by appending the first letter of w, and look up w from $\Phi^{-1}[k]$ again.
4. Write w to the output.
5. Add incomplete w to Φ at index K.
6. $K \leftarrow K + 1$.

Code listing 2.3: The LZW compression algorithm. The algorithm maintains a dynamic dictionary Φ, initialized to contain one entry for each symbol in the alphabet $\mathcal{X}$. A new word is added to Φ in each iteration of the algorithm. The number of words in Φ always equals K, and the words are stored at positions 0 to $K-1$ in Φ.
Notes: The end condition of the decoder's loop in step B could be triggered after decoding the special `EOF` symbol, or after a previously communicated message length is reached.

Many more variants of the above algorithms exist; some published in journals or conference proceedings, others simply as source code, e.g. LZMA. As of 2014, advances in dictionary coding techniques are still regularly published at leading data compression conferences.

2.3.3 Properties

There are some notable properties of dictionary coding algorithms. Firstly, the compression ratio of a dictionary coder will (in the infinite limit) converge to the entropy of the input sequence, assuming an ergodic source. In practice, however, this convergence can be slow. Secondly, they can operate at high speeds, and are not particularly difficult to implement. Dictionary coders have been very successful and are widely used.

However, they do not separate the task of data modelling from the task of encoding. This means that dictionary coders do not provide an explicit understanding of the probabilistic assumptions they make about the input sequence, and offer few opportunities to adapt the

algorithms to new kinds of input data, at least not straightforwardly.

On human language text and source code, dictionary coders are typically outclassed in compression effectiveness by algorithms such as `bzip2`, CSE, CTW and PPM.[4] Of these latter algorithms, `bzip2` is described in section 2.4.2 and PPM is investigated in detail in chapter 6. Figure 2.3 (on page 36) shows the compression effectiveness of dictionary coders in a comparison against other algorithms.

2.4 Transformations

Some classical compression algorithms take a modular approach to compression, where instead of devising a direct coding scheme for the data, the input is transformed into something that is easier to compress using existing algorithms.

2.4.1 Move-to-front encoding

An example of such a technique includes "move-to-front" encoding (Ryabko, 1980; Bentley et al., 1986; Elias, 1987; Willems, 1989), which transforms a sequence of symbols $x_1 \ldots x_N$ into a sequence of natural numbers $k_1 \ldots k_N$, where each k_n is a backwards pointer to the most recent occurrence of x_n. Each k_n is the number of unique symbols that were seen since x_n last occurred.

One way of implementing a move-to-front encoder for a finite alphabet $\mathcal{X}$ is to maintain a recency queue, containing one instance of each symbol x in the alphabet. Each symbol in the sequence is translated to the number indicating the symbol's current position in the queue; after each encoding, the most recently encoded symbol is moved to the front of the queue, shifting the other symbols back. The decoder can recover the original sequence $x_1 \ldots x_N$ from the list of numbers $k_1 \ldots k_N$ by replaying the actions of the encoder, using an exact replica of the encoder's recency queue.

Symbols that occur frequently in the original sequence will, on average, be assigned small numbers in the transformed sequence. The transformed sequence $k_1 \ldots k_N$ can be encoded in a space-efficient way when small numbers map to short code words, e.g. using a Huffman code (for finite $\mathcal{X}$), or an integer code (for infinite $\mathcal{X}$).

For an infinite alphabet $\mathcal{X}$, Bentley et al. describe an encoding scheme that uses a dynamically growing queue which is initially empty. When a new symbol is encountered, a special number is transmitted (e.g. -1) followed by a direct encoding of the new symbol itself.

[4] Compression by substring enumeration (CSE) by Dubé and Beaudoin (2010) and context tree weighting (CTW) by Willems et al. (1993) are not discussed further in this thesis, but results are included for comparison.

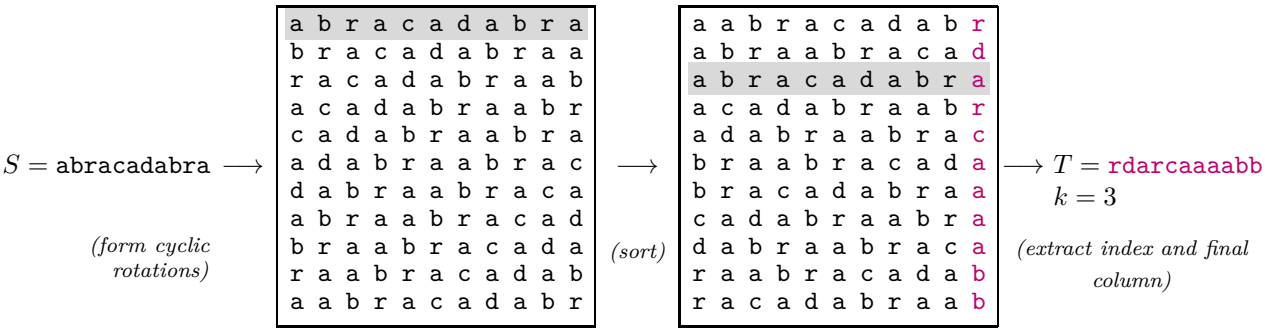


Figure 2.2: An illustrated example of the Burrows–Wheeler transform, showing how the input string S is used to construct the transformed string T and index k.

An example where move-to-front encoding is used in practice is the `bzip2` algorithm, which is explained in the next section.

2.4.2 Block compression

One of the most astonishing classical compression algorithms involves the block-sorting transform of Burrows and Wheeler (1994). The transform itself does not compress its input sequence; it merely reversibly rearranges the sequence to a permutation that ends up being easier to compress.

A notable compressor based on the Burrows–Wheeler block transform is `bzip2` (Seward, 1997–2010), which is included with most contemporary Linux, MacOSX and Unix operating systems.

The transform works as follows. Given a string S (of N symbols), generate the list of all N cyclic rotations of S, and sort this list in lexicographical order. Record the position k at which the original (unrotated) string S appears in this sorted list. The transformed string T is obtained by stepping through the list in sorted order, collecting the *final symbol* of each string, and concatenating the result. The resulting string T has length N and is a particular permutation of the input string S. An example is shown in Figure 2.2.

The Burrows–Wheeler transform (BWT) has some remarkable properties:

(1) the transform is reversible, i.e. knowing only the transformed string T and index k, it is possible to reconstruct the original string S.

(2) the transformed string T has characteristics that make it fairly easy to compress with simple algorithms; for example, `bzip2` uses move-to-front encoding followed by Huffman coding.

Here is an intuitive argument why the transformed string may be easier to compress. A symbol in natural language text is well predicted by the symbols immediately surrounding it. The

symbols in the final column of the matrix are those which in the original string directly precede the symbols in the first column, and the first column is in sorted order. The symbols in the transformed string are therefore arranged in the order of their sorted suffixes, grouping together similar symbols.

Details can be found in the original article by Burrows and Wheeler (1994), additional research and analysis is provided in technical reports by Fenwick (1996) and Balkenhol and Kurtz (1998). An interesting relationship between BWT and PPM is revealed by Cleary et al. (1995, section 4).

2.5 Summary

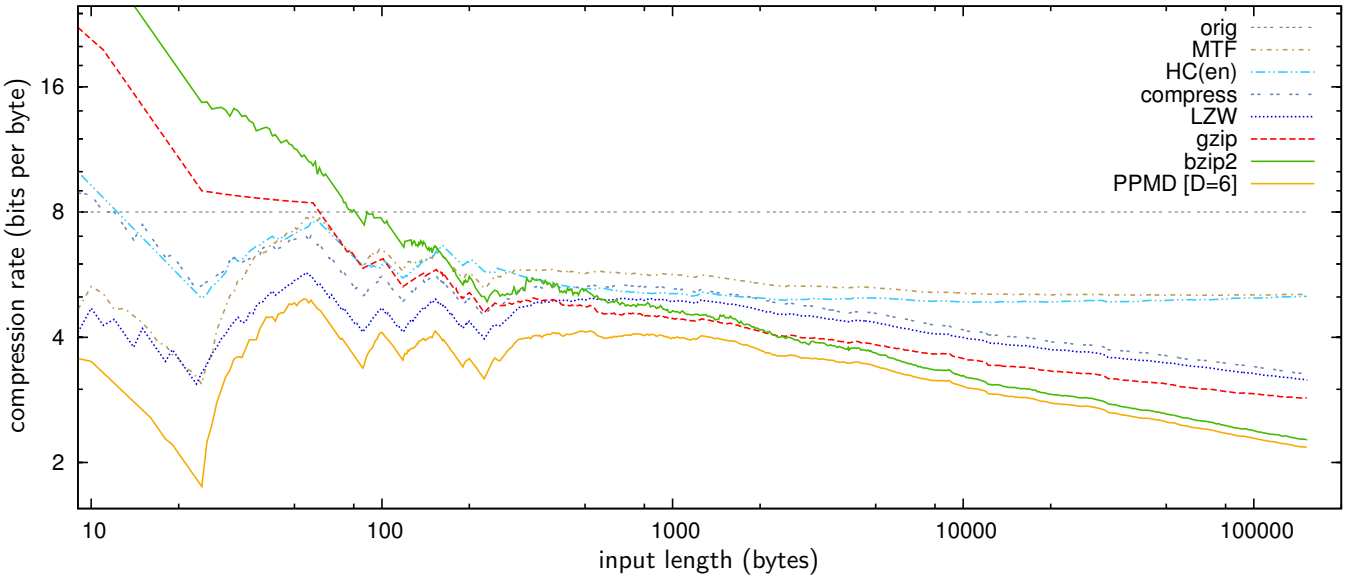
This chapter reviewed a selection of classical compression algorithms. Examples included integer codes (defined over a countably infinite set of numbers), and symbol codes (over a finite alphabet of symbols). The Huffman algorithm computes an optimal symbol code for the nearest dyadic approximation to any finite probability distribution. We also looked at algorithms that capture some contextual dependencies of the input sequence, including dictionary coders and compressors based on the Burrows–Wheeler block transform. Figure 2.3 shows the compression effectiveness of selected algorithms.

The academic discipline of data compression can be regarded as the art of finding codes which convey, on average, an arbitrary input message with a shorter number of symbols than the original input. Each coding scheme necessarily makes assumptions about the data, and compression is only possible with data for which these assumptions are mostly true. The assumptions made by the methods in this chapter may not be obvious from their algorithmic description. One reason for this lack of transparency might be that these algorithms were expressed as an explicit transformation of the input sequence, either by directly producing the final sequence of `1`s and `0`s (e.g. in classic symbol and integer codes), or by producing an intermediate sequence of integers (e.g. dictionary and transform coders) which is then compressed further by another method. The units in which these methods "think" are the output symbols, rather than the probability distributions over inputs their operations implicitly define.

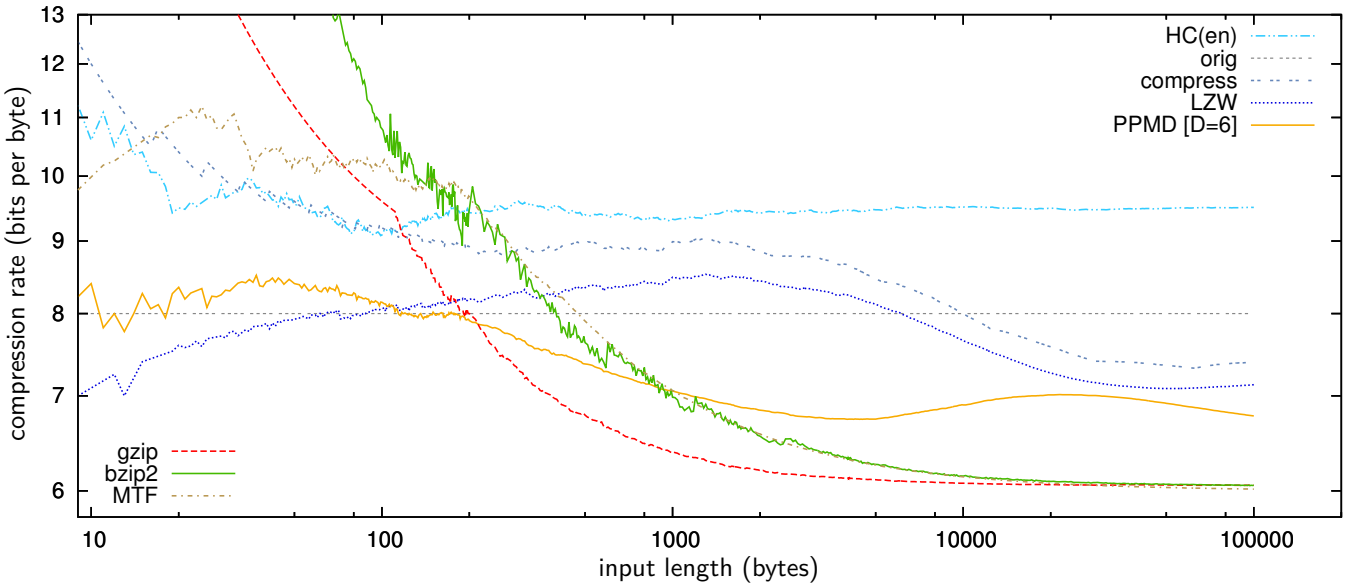
An alternative approach to data compression is to separate the task of coding from the task of data modelling. This separation makes compression methods easier to design and modify, as the assumptions about the data are explicitly represented in the form of probability distributions, and the generation of `1`s and `0`s is delegated to a general compression coding algorithm, such as an arithmetic coder. Arithmetic coders (and similar algorithms) place almost no restrictions on the kinds of probability distributions that can be used. This fortunate circumstance allows compression algorithms to focus mainly on the modelling, i.e. the design and computation of adaptive probability distributions.

The decoupling of coding and modelling is always possible, and never adversely affects compression effectiveness. There may, however, be a cost in terms of computation or algorithmic simplicity in some cases. For example, there are particularly concise and efficient coding algorithms for some restricted classes of probability distributions; the symbol and integer coding methods described in this chapter are instances of such algorithms.

On most contemporary computing equipment, the computational costs of using arithmetic coding are fairly negligible, especially compared to the cost of inference in complex data models.



(a) Compression effectiveness of selected algorithms, measured on English language text "Alice in Wonderland" (`alice29.txt` of the Canterbury corpus). For each input length on the x-axis, a truncated version of the text was made and compressed with all algorithms. The y-axis shows the compressed length (in bits) divided by the input length (in bytes).



(b) The same algorithms measured on a sequence of random symbols (`random.txt` of the artificial corpus). The sequence contains 100 000 symbols, drawn uniformly from a subset of 64 (out of 256 possible) 8-bit symbols. Details on the behaviour of compression algorithms on random sequences can be found in section 8.3.

Figure 2.3: Compression effectiveness as a function of input length, for selected classical algorithms. LZW, compress and gzip are dictionary compressors, bzip2 is a block-sorting compressor, HC(en) is a fixed Huffman code for English text, MTF a move-to-front encoder (with adaptive index compressor), and orig shows the file's original encoding. PPMD is described in chapter 6. (Standard `gzip`, `bzip2` and Unix `compress` were used for this plot; HC(en), MTF, LZW and PPMD are my own implementations. Table A.1 on page 186 contains decriptions of all algorithms.)

Chapter 3

Arithmetic coding

This chapter describes arithmetic coding, an algorithm that produces, given a sequence of input symbols and associated probability distributions, a near-optimally compressed output sequence whose length is within 2 bits of the input sequence's Shannon information content. The appeal of such a method is that the tasks of data modelling and code generation can be cleanly separated, allowing compression methods to be constructed through data modelling.

This chapter proposes an architecture for a modular compression library based on arithmetic coding, which allows compression algorithms to be designed in a coherent, modular and compositional manner.

For concreteness, this chapter includes a complete implementation of an arithmetic coder as JAVA source code, and describes how to interface it to various fundamental probability distributions that serve as building blocks for the compression methods in later chapters.

3.1 Introduction

3.1.1 Information content and information entropy

Data compression and probability theory are fundamentally linked by the relationship of the probability distribution P, and the expected amount of information required to convey a random P-distributed object. Given P, an object x has an *information content* of $h_P(x)$ bits, where:

$$h_P(x) \;=\; \log_2 \frac{1}{P(x)}. \tag{3.1}$$

Averaging over all possible messages $x \in \mathcal{X}$ yields the *expected* information content of the probability distribution P, called its *information entropy*:

$$\mathrm{H}(P) \;=\; \sum_{x \in \mathcal{X}} P(x) \cdot h_P(x) \;=\; \sum_{x \in \mathcal{X}} P(x) \cdot \log_2 \frac{1}{P(x)} \tag{3.2}$$

The information entropy $\mathrm{H}(P)$ of any finite discrete distribution P is highest when P is uniform, i.e. when all elements have equal probability mass. Both information content and information entropy are measured in *bits*, where one bit is the amount of information needed to convey a choice between two equiprobable options. N bits can convey a choice among 2^N equiprobable options, and a fair choice among K equiprobable options has an information content of $\log_2 K$ bits.

The information entropy of a distribution P marks the theoretical limit down to which P-distributed messages can, on average, be losslessly compressed. No code can expect to communicate N messages in fewer than $N \cdot \mathrm{H}(P)$ bits of information, if the messages are random, independent, and distributed according to P.

3.1.2 The relationship of codes and distributions

As mentioned in section 1.2, every compression code necessarily assumes, at least implicitly, a distribution over input objects. Consider a uniquely decodable code C that maps inputs $x \in \mathcal{X}$ to code words $C(x) \in \mathcal{A}^*$, where $\mathcal{A}$ is some finite alphabet. The compression effectiveness of code C (given $|\mathcal{A}|$, the size of the alphabet) is completely determined by the *lengths* of the code words, i.e. by the set of mappings from x to $|C(x)|$.

These code lengths can be interpreted as a probability distribution P_C over input objects:

$$P_C(x) = \frac{1}{Z} \cdot |\mathcal{A}|^{-|C(x)|} \tag{3.3}$$

where Z normalises the probabilities to unity. Code C works best on objects that are distributed according to P_C. However, C might not be the *best* possible code for P_C: it might be a wasteful code whose code words are longer than necessary.

A lossless compression code is *optimal* for a given probability distribution if the expected code length for a random message is minimal. The most effective code word lengths, for a given distribution P, would be:

$$|C_{\mathrm{OPT}}(x)| \;=\; \log_{|\mathcal{A}|} \frac{1}{P(x)} \;=\; \frac{1}{\log_2 |\mathcal{A}|} \cdot h_P(x). \tag{3.4}$$

As these quantities are non-integer, it is not generally possible to find code words that match these optimal lengths exactly. However, it is possible to construct code words whose integer lengths are "close enough" to the optimal lengths to guarantee that the *expected code word length* is within one symbol of the theoretical optimum:

$$\underbrace{\frac{1}{\log_2 |\mathcal{A}|} \cdot \mathrm{H}(P)}_{\text{optimal length}} \;\leq\; \underbrace{\sum_{x \in \mathcal{X}} P(x) \cdot |C(x)|}_{\text{expected length}} \;<\; \underbrace{\frac{1}{\log_2 |\mathcal{A}|} \cdot \mathrm{H}(P) + 1}_{\text{optimal length+1}}. \tag{3.5}$$

For details and proof, see e.g. Cover and Thomas (2006), chapter 5.

3.1.3 Algorithms for building optimal compression codes

One method for constructing an optimal set of code words that satisfies the bound given in (3.5) is the Huffman algorithm (as described in section 2.1.1). A Huffman code for P has an expected code word length that's within one bit of $\mathrm{H}(P)$, the Shannon entropy of P. However, generating a Huffman code requires enumerating all possible messages in advance, which makes its use impractical for large or unbounded sets of messages, such as files of human text.[1]

Fortunately, there are techniques that can compress single messages efficiently without considering all possible messages beforehand. An *arithmetic coding* algorithm (or "arithmetic coder", for short) is an example of such a method. Arithmetic coders progressively construct, for any message x and associated probability distribution P, a single code word whose length is within 2 bits of $h_P(x)$, the Shannon information content of the input. The expected output length of an arithmetic coder is therefore at most 2 bits worse than that of a Huffman code.

An arithmetic coder can be used whenever the following requirements are fulfilled:

- the input message can be broken up into a series of discrete decisions or symbols,
- the probability distribution over messages can be factorised into a product of univariate conditional distributions (one for each symbol or decision),
- the *cumulative distribution* of each of these univariate distributions can be computed.

Section 3.2 describes the principal idea behind arithmetic coding, and includes a concrete implementation of the algorithm.

Huffman coding versus arithmetic coding. An advantage of arithmetic coding is its ability to compute the code word for a message without having to consider all other possible messages or code words. Huffman coding, by comparison, requires comparing the probability mass of all possible messages. Disadvantages of arithmetic coding include that the generated output can be longer than that of a Huffman code. Consider, for example, encoding exactly one of two possible messages $\{\mathtt{A}, \mathtt{B}\}$, where $P(\mathtt{A}) = 0.999$ and $P(\mathtt{B}) = 0.001$. The information contents of the two messages are $h_P(\mathtt{A}) \approx 0.001$ bits and $h_P(\mathtt{B}) \approx 9.966$ bits. The Huffman algorithm allocates code words of length 1 to each message, which is optimal if only one

[1] Note that a Huffman code over *source symbols* is optimal only for encoding a single symbol. Encoding a sequence of symbols by concatenating those Huffman code words is not generally optimal (except when the symbols in the sequence are independent and identically distributed with known and strictly dyadic symbol probabilities).

message is to be transmitted. An arithmetic coder requires 1 bit of output to communicate message `A`, and ca. 10 bits for message `B`.

Assuming that only a single message is being transmitted, and that all messages and their probabilities $P(x)$ can be enumerated, the Huffman algorithm is guaranteed to produce an optimal code for P-distributed messages. If the number of possible messages is very small, and if the probability distribution over messages doesn't change, constructing a Huffman code is most likely the best thing to do; in most other situations (such as compressing files of human text), arithmetic coding is the tool of choice.

Arithmetic coding can be viewed as a natural generalisation of Huffman coding: when used on dyadic input distributions, the output produced by an arithmetic coder is identical to that of a Huffman coder. For a concrete demonstration of the similarity of both algorithms see e.g. the article by Bloom (2010).

A note on the term 'entropy coding'. In the compression literature, there seems to be wide-spread and inconsistent use of the term *entropy coding*. For lack of a precise definition, use of this term is best avoided entirely; more information is given on page 41.

3.2 How an arithmetic coder operates

Arithmetic coding is a general coding method that compresses a sequence of input symbols $x_1 \ldots x_N$, given associated probability distributions $P_1 \ldots P_N$, to a sequence of binary digits whose length is within 2 bits of the sequence's information content. The probability distributions $P_1 \ldots P_N$ must be available to both the sender and the receiver.

In a nutshell, arithmetic coding successively maps each input symbol x_n to an interval $R_n \subseteq R_{n-1}$ in proportion to its probability mass $P_n(x_n)$, starting from $R_0 = [0, 1)$. The encoded file is the binary representation of the shortest number $r \in \mathbb{R}$ inside the final interval R_N.

3.2.1 Mapping a sequence of symbols to nested regions

Consider a sequence of symbols $x_1 \ldots x_N$ with associated probability distributions $P_1 \ldots P_N$. The joint probability mass of the sequence is:

$$\Pr(x_1 \ldots x_N \mid P_1 \ldots P_N) \;=\; \prod_{n=1}^{N} P_n(x_n) \tag{3.6}$$

An arithmetic coder can map this sequence to a series of nested regions of the unit interval, one symbol at a time. The process starts out with the interval $R_0 = [0, 1)$. For some arbitrary ordering of the symbols in the alphabet $\mathcal{X}$, this interval is partitioned into non-overlapping

Reasons for avoiding the term 'entropy coding'

The first occurrence of the term entropy coding I could find appears in a paper by Goblick and Holsinger (1967) on the digitization of analog signals. The authors give a definition with local scope, in the context of a particular encoding method.
An alternative source of the term might be a paper by O'Neal (1967) on the same topic, which contains the following statement:

> [...] *Therefore, the technique of entropy coding* [5] *(also called "Shannon–Fano coding" or "Huffman coding") can be used either to increase the S/N ratio for a given bit rate or to decrease the bit rate for a given S/N ratio.* [...] where [5] refers to a book by Fano (1961).

In 1971, the same author published an article with "entropy coding" in the paper title. The text contains the plural: *"entropy coding techniques (Huffman or Shannon–Fano coding)"*. No reference or definition is given.
A paper by Berger et al. (1972) makes heavy use of the term "entropy coding", but gives no explicit reference to its definition or origin. However, the authors were aware of the paper by Goblick and Holsinger (1967), as it is referenced in their bibliography. From 1972 onwards, many papers contain the term "entropy coding", possibly copying O'Neal's use of the term.
Where definitions do show up, they are rather vague. For example, ITU recommendation H.82 (ITU-T, 1993) defines an "entropy coder" to be *any lossless method for compressing or decompressing data*. Similarly, the book by Wiegand and Schwarz (2011) defines entropy coding to be a synonym for "lossless coding" and "noiseless coding". No explanations or references are given.

What the term "entropy coding" could mean:

- *"any coding method whose expected rate of compression equals the entropy of the input distribution."* But a better criterion than the expected rate of compression is the actual compression: a coding method whose output length is equal to the Shannon information content of the input. Also, it's possible to construct optimal codes for sources that do not even have a finite entropy.

- *"a coding method whose distribution over* output symbols *has high entropy."* Such a definition would not even imply compression; typical encryption methods, for example, also produce output symbols with uniform probability.

In summary, "entropy coding" doesn't seem a clearly defined or distinguished concept. Leading text books, e.g. by Cover and Thomas (1991, 2006) or MacKay (2003), do not contain the term "entropy coding", and discuss optimal codes instead. Following their example, the term entropy coding is not used further in this thesis.

regions, one for each symbol $x \in \mathcal{X}$, in proportion to the mass $P_1(x)$ of its symbol. Once the arithmetic coder knows the first symbol x_1, it chooses the region R_1 matching that symbol, and discards all information about the other regions.

Similarly, the process continues for the remaining symbols. At step n, the previously computed region R_{n-1} is divided into subintervals according to P_n, and the next region R_n is selected by the next symbol x_n.

The final region R_N contains the sequence of choices made for each symbol, such that the size of R_N matches the joint probability mass of all symbols in the sequence, as given in equation (3.6). The key observation is that knowing the final region R_N and the probability distributions P_1 ... P_N is sufficient for reconstructing the original sequence x_1 ... x_N.

3.2.2 Mapping a region to a sequence of symbols

Given the final region R_N and the symbol distributions P_1 ... P_N, the original sequence x_1 ... x_N can be recovered as follows. Starting from $n = 1$ and the unit interval $[0, 1)$, each symbol x_n can be recovered by partitioning the current interval into regions according to P_n, and selecting the region that contains R_N as a subregion. The label of the chosen region is the original symbol x_n. The process continues recursively from the previously chosen region, until all N symbols are recovered.

For the recovery process to work, only a single point (or subregion) inside the final region R_N needs to be known, rather than the exact region boundaries of R_N. In this case, the first N symbols can still be reconstructed perfectly, but N itself cannot. Unless the sequence length N is known in advance, it must be encoded separately, e.g. by using a special `EOF` symbol to mark the termination of the sequence. (Methods of encoding sequence termination are discussed in section 5.7.2.)

Adaptive distributions. Arithmetic coding allows a different distribution to be used for each symbol in the sequence. Furthermore, the probability distribution P_n at timestep n is allowed to depend on the previous symbols x_1 ... x_{n-1}, because P_n is only needed after x_{n-1} has been decoded. This property makes it possible to use adaptive distributions, e.g. distributions that gradually adjust by learning from previous symbols in the sequence. Adaptive compression techniques are described in more detail in chapter 4, and find use in state-of-the-art sequence compressors (such as those investigated in chapter 6).

3.2.3 Encoding a region as a sequence of binary digits

Having shown how to represent a sequence of symbols with a region R_N, and how to recover the sequence again from this region, it remains to be shown how to *encode* R_N to an output

sequence of binary digits. The method is beautiful and simple: we can apply the procedure from section 3.2.2, but instead of using the input alphabet $\mathcal{X}$ and distributions P_1 ... P_N, we use the binary alphabet $\{\mathtt{0}, \mathtt{1}\}$ and uniform probabilities $U(\mathtt{0}) = U(\mathtt{1}) = \frac{1}{2}$ for each subregion.

At every step of this procedure, the current region (again, starting from $[0, 1)$) is subdivided into two equal halves, one for digit `0`, and one for digit `1`. If the known target region R_N lies completely within the `0`-region, a `0` is written, and the `0`-region is used as the enclosing region for the next coding step. Similarly, when R_N lies within the `1`-region, a `1` is written. The procedure repeats until the current region lies completely inside the target region R_N, resulting in M binary output digits s_1 ... s_M. It is then possible to reconstruct a subregion of R_N from s_1 ... s_M (using U_1 ... U_M), and from there also the original sequence x_1 ... x_N (using P_1 ... P_N).

More generally, arithmetic coding can be used to compactly translate a sequence of values from a *source distribution* to a sequence of values distributed according to a *target distribution*. (This slightly more general functionality is used e.g. in section 8.1 to compress messages to output alphabets with unequal symbol lengths.)

Practical arithmetic coding algorithms can perform this translation gradually, keeping only as much state information as is required to produce the next output symbol.

3.3 Concrete implementation of arithmetic coding

In practice, arithmetic coders are typically implemented using fixed width integer arithmetic. A concrete implementation of such a fixed-precision coder was published by Witten, Neal and Cleary (1987), and developed further by Moffat, Neal and Witten (1998). The arithmetic coder proposed in this chapter is based on their work; its full source code is shown on pages 47–48.

Background. In a fixed-precision arithmetic coder, the interval $[0, 1)$ is represented by a range of integers from 0 to $2^{\mathtt{b}} - 1$, where `b` is the bitwidth of available integer variables. The coder for this chapter was written in JAVA and uses variables of type `long` (64 bits wide).

The internal state of an arithmetic coder is the current region R_n, which in a fixed-precision implementation is represented with two integers, e.g. `L` (the low pointer) and `R` (the region width). Fixed-width coders typically also store an additional quantity named `bits_waiting`, which is explained in detail by Moffat et al. (1998). These internal quantities are modified whenever a new source symbol is read from the input, and whenever a compressed symbol is written to the output.

Innovations of the proposed coder. When compressing a sequence with an arithmetic coder, every input symbol x_n must be accompanied by a distribution P_n; this distribution is

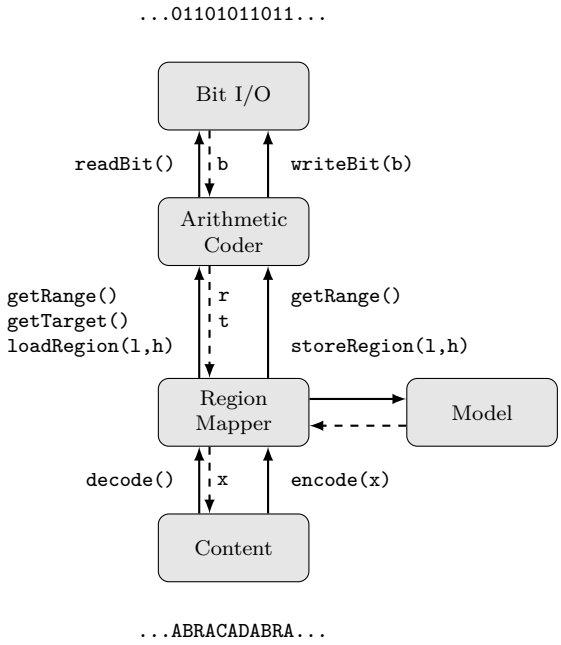


Figure 3.1: Proposed architecture of compression algorithms based on probabilistic models and arithmetic coding. The *region mapper* uses the model's predictive distribution to translate encoding and decoding requests into calls to the arithmetic coder. The components may be separate or combined entities.

used to partition the current region into subregions. The arithmetic coder proposed in this chapter never handles symbols or probabilities explicitly, and instead relies on the implementation of the probabilistic source model to translate symbols x_n and symbol distributions P_n to discrete regions.

This design choice allows probability distributions to offer a simple function interface for compressing and decompressing values, without the need to expose the inner workings of the arithmetic coder. For example, encoding a value `x` with a distribution P can be expressed as P`.encode(x,ec)`, where `ec` is an instance of an arithmetic *encoder*. Similarly, the corresponding inverse operation is given by `x =` P`.decode(dc)` where `dc` is an instance of an arithmetic *decoder*. P need not be a distribution over symbols, it could be a distribution over numbers, vectors, strings, sets or any type of object. Figure 3.1 shows a graphical representation of the proposed interface.

The implementation of the probability distribution P must provide the `encode` and `decode` functions: these two functions are responsible for mapping any element x to a discrete region whose size is approximately proportional to $P(x)$.[2] The region boundaries are communicated to an arithmetic coder instance using the functions shown in Table 3.1.

[2]At least implicitly, the `encode` and `decode` functions necessarily compute a discrete form of P_Σ (the cumulative distribution of P). The implementation must ensure that every input x with positive probability mass $P(x) > 0$ is mapped to a region of non-zero width, as those inputs would otherwise become unencodable.

Use	Operation	Description
ENC+DEC	`long getRange()`	Returns `R`, the current range of the coder.
ENC	`void storeRegion(l,h)`	Zooms into the discrete region `(l,h)`, and writes compressed output bits if possible. The region is specified with integers `l`, the lower (inclusive) point of the region, and `h`, the upper (exclusive) point.
DEC	`long getTarget()`	Returns an integer between 0 (inclusive) and `R` − `L` (exclusive) that allows the decoder to identify the next region.
DEC	`void loadRegion(l,h)`	Zooms into the discrete region `(l,h)`, and reads compressed bits from the input if necessary.

Table 3.1: Components of the arithmetic encoding and decoding interface. Probabilistic model implementations use these functions to communicate discrete regions for encoding and decoding. Code listing 3.5 shows how a model's `encode` and `decode` methods might be structured.

The design of suitable `encode` and `decode` functions is not always trivial; for example, multivariate distributions and distributions with infinite support may require special treatment. A selection of useful techniques can be found in section 3.4. Implementations for many common probability distributions are included in the compression library written for this thesis.

A historically prominent application of arithmetic coding is the encoding of symbols whose probability mass is computed from their empirical occurrence counts, for example as follows:

$$P(x_N = x) = \frac{\#[x_n = x]_{n=1}^{N-1} + 1}{N + |\mathcal{X}|}. \tag{3.7}$$

where $|\mathcal{X}|$ is the size of the symbol alphabet. This application presumably motivated the design of the procedure `narrow_region(l,h,t)` in the Witten–Neal–Cleary coder, which computes the next region by scaling cumulative counts `l` and `h` relative to a total count `t`.

Of course, adaptive symbol models are not limited to the form shown in (3.7), and symbol probabilities are not generally integer fractions. For these reasons, I advocate the more general interface shown in Table 3.1, as it allows models greater control over the computation of regions. For completeness and convenience, traditional versions of `storeRegion`, `loadRegion` and `getTarget` that include built-in scaling are given in code listing 3.4.[3]

[3]The original form of the PPM algorithm by Cleary and Witten (1984a) can be implemented entirely using these traditional versions. An in-depth treatment of PPM is given in chapter 6.

Arithmetic Coding: example application

```
public class ABC {

  /** Local state */
  int na = 1;
  int nb = 1;
  int nc = 1;
  int all = 3;

  /** Records a symbol observation. */
  public void learn(char c) {
    switch(c) {
      case 'A': na++; break;
      case 'B': nb++; break;
      case 'C': nc++; break;
      default : throw new IllegalArgumentException();
    }
    all++;
  }

  /** Encodes a symbol. */
  public void encode(char c, Encoder ec) {
    switch(c) {
      case 'A': ec.storeRegion(0, na, all); break;
      case 'B': ec.storeRegion(na, na+nb, all); break;
      case 'C': ec.storeRegion(na+nb, all, all); break;
      default : throw new IllegalArgumentException();
    }
  }

  /** Decodes a symbol. */
  public char decode(Decoder dc) {
    long t = dc.getTarget(all);
    if (t >= 0) {
      if (t < na) {
        dc.loadRegion(0, na, all);
        return 'A';
      } else
      if (t < na+nb) {
        dc.loadRegion(na, na+nb, all);
        return 'B';
      } else
      if (t < all) {
        dc.loadRegion(na+nb, all, all);
        return 'C';
      }
    }
    throw new IllegalStateException();
  }

} // end of class ABC
```

```
public class ABCTest {

    public static void main(String[] args) throws Exception {
      /* Compressing a sequence of ternary values */
      char[] data = new char[] { 'B','B','B','B','A','B','B','B','C' };
      ABC abc = new ABC();
      Arith ac = new Arith();
      BitWriter bw = IOTools.getBitWriter("output.bin");
      ac.start_encode(bw);
      for (char x : data) {
        abc.encode(x,ac);
        abc.learn(x);
      }
      ac.finish_encode(); // writes "0111 1001 0100 1"
      bw.close();         // appends "000" to fill the last byte

      /* Decompressing the sequence */
      abc = new ABC();  ac = new Arith();
      BitReader br = IOTools.getBitReader("output.bin");
      ac.start_decode(br);
      String s=""; // we append decompressed symbols here
      char x;
      do {
        x = abc.decode(ac);
        abc.learn(x);
        s += x;  // append symbol x
      } while (x != 'C');
      ac.finish_decode();
      br.close();
      System.out.println("Decoded: "+s); // prints "BBBBABBBC"
    }

} // end of class ABCTest
```

Code listing 3.1: An example of arithmetic coding used to compress a sequence of ternary values $x_n \in \{\texttt{A}, \texttt{B}, \texttt{C}\}$. The class `ABC` implements three methods: `encode`, `decode`, and `learn`. These methods are called in the class `ABCTest` for adaptively compressing a fixed sequence to an external file `output.bin`, and then decompressing this file to recover the original sequence.

Arithmetic Coding

CONSTANTS

```
/** Number of bits available. */
final long b = Long.SIZE - 2;
/** Index of lower quarter. */
final long lb = (long) 1 << (b-2);
/** Index of midpoint. */
final long hb = (long) 1 << (b-1);
/** Index of top point. */
final long tb = ((long) 1 << b) - 1;
/** Mask of b 1-bits. */
final long mask = tb;
```

LOCAL STATE

```
/** Current range of coding interval. */
long R;
/** Low index of coding interval. */
long L;
/** Target location in coding interval. */
long D;             // Decoder only
/** Number of opposite-valued bits queued. */
long bits_waiting; // Encoder only

BitWriter output = null;
BitReader  input = null;
```

MAIN PROCEDURES

```
/** Outputs encoder's processed bits. */
void output_bits() {
  while (R <= lb) {
    if (L+R <= hb) {
      output_all((byte) 0);
    } else
    if (L >= hb) {
      output_all((byte) 1);
      L = L - hb;
    } else {
      bits_waiting++;
      L = L - lb;
    }
    L <<= 1;  R <<= 1;  // zoom in
  }
}

/** Writes a bit, followed by bits_waiting
  * bits of opposite value. */
void output_all(byte bit) {
  output.writeBit(bit);
  while (bits_waiting > 0) {
    output.writeBit((byte) (1 - bit));
    bits_waiting--;
  }
}

/** Sets a region. */
void narrow_region(long l, long h) {
  L = L + l;   // CAUTION: l, not 1
  R = h - l;
}
```

```
/** Discards decoder's processed bits. */
void discard_bits() {
  while (R <= lb) {
    if (L >= hb) {
      L -= hb; D -= hb;
    } else
    if (L+R <= hb) {
      // in lower half: nothing to do
    } else {
      L -= lb; D -= lb;
    }
    L <<= 1; R <<= 1;  // zoom in
    D <<= 1; D &= mask; D += input.readBit();
  }
}
```

```
/** Loads a region. */
public void loadRegion(long l, long h) {
  narrow_region(l,h);
  discard_bits();
}
```

```
/** Returns a target pointer. */
public long getTarget() { return D-L; }
```

```
/** Returns the coding range. */
public long getRange() { return R; }
```

```
/** Encodes a region. */
public void storeRegion(long l, long h) {
  narrow_region(l,h);
  output_bits();
}
```

Code listing 3.2: JAVA source code of the main operational procedures in an arithmetic coder. The shaded methods are intended for external use.

Arithmetic Coding: starting and stopping

STARTING

```
/** Starts an encoding process. */
void start_encode(BitWriter output) {
  this.output = output;
  L = 0;   // lowest possible point
  R = tb;  // full range
  bits_waiting = 0;
}
```

```
/** Starts a decoding process. */
void start_decode(BitReader input) {
  this.input = input;
  D = 0;
  // fill data pointer with bits
  for (int k=0; k<b; k++) {
    D <<= 1;
    D += input.readBit();
  }
  L = 0;
  R = tb; // WNC use "tb", MNW use "hb"
}
```

STOPPING

```
/** Finishes an encoding process. */
void finish_encode() {
  while (true) {
    if (L + (R>>1) >= hb) {
      output_all((byte) 1);
      if (L < hb) {
        R -= hb - L;   L  = 0;
      } else {
        L -= hb;
      }
    } else {
      output_all((byte) 0);
      if (L+R > hb) { R = hb - L; }
    }
    if (R == hb) { break; }
    L <<= 1;  R <<= 1;
  }
}
```

```
/** Finishes a decoding process. */
void finish_decode() {
  // no action required
}
```

Code listing 3.3: JAVA source code of the start and stop procedures in an arithmetic coder.

Arithmetic Coding: default scaling

```
void narrow_region(long l, long h, long t) {
  long r = R / t;
  L = L + r*l;
  R = h < t ? r * (h-l) : R - r*l;
} // Moffat-Neal-Witten (1998)
```

```
void storeRegion(long l, long h, long t) {
  narrow_region(l,h,t);
  output_bits();
}
```

```
void loadRegion(long l, long h, long t) {
  narrow_region(l,h,t);
  discard_bits();
}
```

```
long getTarget(long t) {
  long r = R / t;
  long dr = (D-L) / r;
  return (t-1 < dr) ? t-1 : dr;
} // Moffat-Neal-Witten (1998)
```

```
void narrow_region2(long l, long h, long t) {
  long T = (R*l) / t;
  L = L + T;
  R = (R*h) / t - T;
} // Witten-Neal-Cleary (1987)

long getTarget2(long t) {
  return (((D-L+1)*t)-1) / R;
} // Witten-Neal-Cleary (1987)
```

Code listing 3.4: JAVA source code for the "default scaling" variants of the main interface methods `storeRegion`, `loadRegion` and `getTarget`, using the scaling code by Moffat, Neal and Witten (1998). The scaling method by Witten, Neal and Cleary (1987) is included for comparison.

Template methods for arithmetic encoding and decoding

COMPRESSION

```
void encode(X x, Encoder ec) {
  long r = ec.getRange();
  // find (l,h) for x, given r
  ec.storeRegion(l,h);
}
```

DECOMPRESSION

```
X decode(Decoder dc) {
  long r = dc.getRange();
  long t = dc.getTarget();
  // find (l,h,x) from (r,t)
  dc.loadRegion(l,h);
  return x;
}
```

Code listing 3.5: A sketch of low-level `encode` and `decode` methods implementing compression and decompression using the arithmetic coding interface. The implementing class would typically be a probability distribution over objects `x` of a class `X`, and include means of computing discrete regions `(l,h)` for any object `x` (given the currently available coding range `r`).

3.3.1 Historical notes

Arithmetic codes were first made practical through the development of coding operations in finite precision arithmetic by Rissanen (1976) and Pasco (1976). These ideas were advanced further by Rissanen and Langdon (1979, 1981); Langdon (1984) and others, but it is fair to add that many people worked on arithmetic codes, with the original idea of the method even tracing back to Shannon's paper on information theory (1948). Details on the history of arithmetic coding can be found in the PhD thesis of Sayir (1999).

The arithmetic coding algorithm of this chapter is based on the designs by Witten, Neal and Cleary (1987) and Moffat, Neal and Witten (1998). A tutorial on arithmetic coding can be found e.g. in a technical report by Howard and Vitter (1992).

Aside. Arithmetic coding can be understood as a generalised change of number base. As an illustration, consider feeding e.g. a few hundred digits of the binary expansion of $\frac{\pi}{10} =$`.010100 000110110010111101110110100111001 1`... into the arithmetic decoder described earlier. The decoder, decompressing this input sequence to uniformly distributed decimal digits {`0...9`}, produces the familiar looking output sequence `3141592653589793`2`414227`... The shaded region indicates where the decoded digits begin to differ from the true decimal expansion of π: this deviation is a consequence of using finite-precision arithmetic coding.

3.3.2 Other generic compression algorithms

There are other generic algorithms that produce near-optimal compression codes for arbitrary distributions. The *Q-coder* by Pennebaker et al. (1988), for example, is a carefully optimised

implementation of arithmetic coding for binary alphabets. It is always possible, given a discrete probability distribution and an outcome x, to encode x as a sequence of biased binary choices; the Q-coder can therefore be used as universally as an arithmetic coder can.

Similarly, the *Z-coder* by Bottou et al. (1998) produces an optimal compression code for any sequence of binary input symbols and associated biases. The operation of the Z-coder is different from the Q-coder and the arithmetic coding algorithm, and is based on a generalisation of Golomb codes (Golomb, 1966).

3.4 Arithmetic coding for various distributions

In this section, I'll discuss how arithmetic coding can be used in practice to encode data from various basic probability distributions. These basic distributions provide important building blocks from which more complex models can be built. Examples of such models can be found in later chapters of this thesis. The techniques presented here are therefore of important practical use, and also serve as concrete examples of how arithmetic coding can be applied in practice.

3.4.1 Bernoulli code

The perhaps simplest use of an arithmetic coder is the compression of Bernoulli variables with known bias, i.e. a sequence of binary outcomes from coin flips with a known bias φ. Of course, such a sequence could be encoded naïvely by simply writing the outcomes with binary symbols $\in \{0, 1\}$ unmodified, and in the case that the coin is fair (i.e. $\varphi = \frac{1}{2}$), this encoding procedure is in fact optimal. But for biases different from $\frac{1}{2}$, or output alphabets other than binary, an arithmetic coder will be more effective.

The Bernoulli distribution over values $\{0, 1\}$ is defined as:

$$\text{Bernoulli}(x \mid \varphi) \;=\; \varphi^{1-x}\,(1-\varphi)^{x} \tag{3.8}$$

where the bias φ is the probability of obtaining a 0. To encode a Bernoulli-distributed value x with an arithmetic coder, each possible value (0 or 1) must be mapped to a region whose size is roughly proportional to its probability. Using the interface from Table 3.1, this mapping can be implemented by checking the coder's current range R with `getRange()`, and finding the integer M that is closest to $R \cdot \varphi$, and not equal to 0 or R itself. The tuples $(0, M)$ and (M, R) then identify the regions. Concrete procedures for encoding and decoding are shown in code listing 3.6.

Arithmetic coding for Bernoulli distributions

ENCODING

`void encode(int x, Encoder ec) {`

1. Get $R \leftarrow$ `ec.getRange()`.
2. Compute $M \leftarrow$ `round`$(R \cdot \varphi)$.
3. Ensure $0 < M < R$.
   (If necessary, adjust $M \leftarrow M \pm 1$.)
4. If $x = 0$:
   `ec.storeRegion`$(0, M)$.
   Otherwise:
   `ec.storeRegion`(M, R).

}

DECODING

`int decode(Decoder dc) {`

1. Get $R \leftarrow$ `dc.getRange()`.
2. Compute $M \leftarrow$ `round`$(R \cdot \varphi)$.
3. Ensure $0 < M < R$.
   (If necessary, adjust $M \leftarrow M \pm 1$.)
4. Set $T \leftarrow$ `dc.getTarget()`.
5. If $T \leq M$:
   `dc.loadRegion`$(0, M)$. Return 0.
   Otherwise:
   `dc.loadRegion`(M, R). Return 1.

}

Code listing 3.6: Fixed-precision arithmetic encoding and decoding procedures for a Bernoulli distribution with bias φ, where φ is the probability of outcome 0. The arguments `ec` and `dc` are pointers to instances of an arithmetic encoder and decoder.

3.4.2 Discrete uniform code

Generalising a fair choice between two options, a discrete uniform distribution defines a choice among N equiprobable options (for any finite N), such that each value $n \in \{1, ..., N\}$ has probability $\frac{1}{N}$. Encoding and decoding procedures for such a distribution could use the scaling versions of `storeRegion`, `loadRegion`, and `getTarget` from code listing 3.4. Encoding an integer n is as simple as invoking `ec.storeRegion`$(n-1,\ n,\ N)$. Decoding works by obtaining $k \leftarrow$ `dc.getTarget`(N), calling `dc.loadRegion`$(k,\ k+1,\ N)$, and returning $n = k+1$.

3.4.3 Finite discrete distributions

The approach of the previous two sections can be generalised to distributions P over N possible values, where N is finite. Each of the N possible values $\{1, ..., N\}$ must be mapped to a discrete region in the coder's current range R.

Any method that achieves the above necessarily computes the P's *cumulative distribution* in some form. Intuitively, this is because each element is mapped to a line-segment of a length proportional to the element's probability mass, and the location of the segment is given by the sum of all preceding segment sizes. The order of the segments doesn't matter, and can be chosen for computational convenience, as long as the encoder and decoder agree.

P's cumulative distribution can be defined as follows:

$$P_\Sigma(n) \stackrel{\text{def}}{=} \sum_{k=1}^{n-1} P(k) \tag{3.9}$$

$$\text{and} \quad P^+_\Sigma(n) \stackrel{\text{def}}{=} P_\Sigma(n) + P(n). \tag{3.10}$$

The cumulative distribution can be used to map each $n \in \{1, ..., N\}$ to a region $[P_\Sigma(n), P^+_\Sigma(n))$ of length $P(n)$ inside the unit interval. To use these regions with our fixed-precision arithmetic coder, the boundaries must be scaled to integers between 0 and R, taking care that there are no gaps and that no region has a width of zero.

A brute-force way of computing discrete regions with the above constraints is shown in code listing 3.7. To improve efficiency, it may help to choose an ordering in which high probability elements (i.e. large regions) come first, so that the expected number of summations and evaluations of P is kept low.

The computational costs can be lowered in certain circumstances. For example, when the distribution P is used frequently and changes rarely, one might cash a precomputed copy of the cumulative distribution P_Σ and scale it on demand. Even cheaper alternatives may exist when P_Σ can computed in closed form, avoiding the need to iterate over the elements.

3.4.4 Binomial code

The binomial distribution describes the number of successes versus failures in a set of N independent Bernoulli trials. It is parametrised by natural number N and success probability θ, and ranges over positive integers $n \in \{0 \dots N\}$. A binomial random variable has the following probability mass function:

$$\text{Binomial}(n \mid N, \theta) = \binom{N}{n} \cdot \theta^n \ (1-\theta)^{N-n} \tag{3.11}$$

Encoding a binomial random variable with an arithmetic coder requires computing the cumulative distribution function of the binomial distribution. A method for doing this efficiently is to make use of the following recurrence relation:

$$\text{Binomial}(n+1 \mid N, \theta) = \frac{N-n}{n+1} \cdot \frac{\theta}{1-\theta} \cdot \text{Binomial}(n \mid N, \theta) \tag{3.12}$$

The cumulative binomial distribution can then be computed as follows. Initialise:

$$B_\Sigma := 0 \tag{3.13}$$

$$B := (1-\theta)^N \tag{3.14}$$

Budget allocation algorithm

```
public static <X> HashMap<X,Long> getDiscreteMass(Mass<X> mass,
                                                  Iterable<X> set, long budget) {
  HashMap<X,Long> map = new HashMap<X,Long>();
  int n = 0; // number of elements
  long sum = 0L; // total allocated mass
  for (X x : set) {
    final double m = mass.mass(x);
    long mb = (long) (m * budget);
    if (mb <= 0L) { mb = 1L; }
    map.put(x, mb);
    sum += mb; n++;
  }
  // deal with left-over or overspent budget:
  long diff = budget - sum;
  if (diff > 0L) {
    // underspent budget
    int delta =
          (diff > n) ?  (int) (diff/n) : 1;
    for (Map.Entry<X,Long> e : map.entrySet()) {
      e.setValue(e.getValue()+delta);
      diff-=delta;
      if (diff==0L) { break; }
      else
      if (diff <= n) { delta = 1; }
    }
  } else
  if (diff < 0L) {
    // overspent budget
    int delta =
          (diff < -n) ? (int) - (diff/n) : 1;
    for (Map.Entry<X,Long> e : map.entrySet()) {
      Long v = e.getValue();
      if (v > 1L) {
        e.setValue(v-delta);
        diff+=delta;
      }
      if (diff==0L) { break; }
      else
      if (diff >= -n) { delta = 1; }
    }
  }
  return map;
}
```

```
public static <X> void encode(X x,
                    Mass<X> mass,
                    Iterable<X> order,
                    Encoder ec) {
  long sum = 0L;
  Map<X,Long> map = getDiscreteMass(
     mass, set, order, ec.getRange());
  for (X y : order) {
    long m = map.get(y);
    if (x.equals(y)) {
      ec.storeRegion(sum,sum+m);
      return;
    }
    sum += m;
  }
  // unsupported element
  throw new RuntimeException();
}
```

```
public static <X> X decode(
                      Mass<X> mass,
                      Iterable<X> order,
                      Decoder dc) {
  long sum = 0L;
  Map<X,Long> map = getDiscreteMass(
     mass, set, order, dc.getRange());
  long r = dc.getTarget();
  for (X x : order) {
    long m = map.get(x);
    if (r >= sum && r < sum+m) {
      dc.loadRegion(sum,sum+m);
      return x;
    }
    sum += m;
  }
  // unused coding range
  throw new RuntimeException();
}
```

Code listing 3.7: A JAVA method that discretizes a probability mass function `mass` (over any iterable space `X`) to a given integer budget. The method returns a map from elements in `X` to integers that sum to the given budget. Summing the resulting integers returns region boundaries that can be used with the arithmetic coding interface: suitable implementations of `encode` and `decode` are shown in the shaded boxes.

Note: These three methods are presented in this form mainly for clarity of illustration, they can (and should) be optimised.

To encode a binomially distributed value n, repeat for each k from $1 \ldots n$:

$$B_\Sigma := B_\Sigma + B \tag{3.15}$$

$$B := \frac{N-k}{k+1} \cdot \frac{\theta}{1-\theta} \cdot B \tag{3.16}$$

The interval $[B_\Sigma, B_\Sigma + B)$ is then a representation of n that can be scaled for use with a finite-precision arithmetic coder.

3.4.5 Beta-binomial code

The Beta-binomial compound distribution results from integrating out the success parameter θ of a binomial distribution, assuming θ is Beta distributed. The Beta distribution is a continuous distribution over the unit interval $[0, 1]$, and is defined as follows:

$$\mathrm{Beta}(\theta \mid \alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\,\Gamma(\beta)} \cdot \theta^{\alpha-1}\,(1-\theta)^{\beta-1}\,, \tag{3.17}$$

where $\Gamma(\cdot)$ denotes the Gamma function.[4] The Beta-binomial distribution expresses the probability of getting exactly n heads from N coin flips, when the coin has an unknown, Beta-distributed bias θ.

The Beta-binomial distribution is parametrised by the number of trials N and the parameters α and β of the Beta prior:

$$\mathrm{BetaBin}(n \mid N, \alpha, \beta) = \int \mathrm{Binomial}(n \mid N, \theta) \cdot \mathrm{Beta}(\theta \mid \alpha, \beta)\, d\theta \tag{3.18}$$

$$= \binom{N}{n} \cdot \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\,\Gamma(\beta)} \cdot \frac{\Gamma(\alpha+n)\,\Gamma(\beta+N-n)}{\Gamma(\alpha+\beta+N)} \tag{3.19}$$

Just like for the binomial distribution, there is a recurrence relation that can be used to compute the probabilities for successive values of n:

$$\mathrm{BetaBin}(n+1 \mid N, \alpha, \beta) = \frac{N-n}{n+1} \cdot \frac{\alpha+n}{\beta+N-n-1} \cdot \mathrm{BetaBin}(n \mid N, \alpha, \beta) \tag{3.20}$$

Summing these probabilities in ascending order of n yields the cumulative distribution. The method described in section 3.4.4 can be modified accordingly, yielding a Beta-binomial coding scheme.

Binomial and Beta-binomial codes are used e.g. in chapter 7 as part of a compressor for multisets of binary sequences.

[4] The Gamma function can be regarded a continuous generalisation of the factorial function, with $k! = \Gamma(k+1) = k \cdot \Gamma(k)$. The Beta distribution gets its name from the Beta function $\mathrm{B}(\alpha, \beta) = \Gamma(\alpha)\Gamma(\beta)/\Gamma(\alpha+\beta)$, as $\mathrm{B}(\alpha, \beta)^{-1}$ is the normalising constant of the Beta distribution.

3.4.6 Multinomial code

The K-dimensional *multinomial distribution* is a generalisation of the binomial distribution to trials with K possible outcomes. It is parametrized by the number of trials N, and by a discrete probability distribution Q over the K possible values. A draw from a multinomial distribution is a K-dimensional vector $\boldsymbol{n} = (n_1 \ldots n_K)$ over positive integers, whose components sum to N. The probability mass function of the multinomial distribution is:

$$\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) = N! \prod_{k=1}^{K} \frac{Q(k)^{n_k}}{n_k!} \tag{3.21}$$

Interfacing a multinomial distribution to an arithmetic coder is not as straightforward as with distributions over integers, because the coding domain is vector-valued and may have a large state space. Each possible vector must map to a coding region, which requires fixing some total ordering over vectors. The approach recommended here is to encode $\boldsymbol{n}$ by decomposing it into a series of simpler coding steps, preserving optimality.

The technique described here exploits a decomposition property of the Multinomial distribution:

$$N! \prod_{k=1}^{K} \frac{Q(k)^{n_k}}{n_k!} = \left((N-n_1)! \cdot \prod_{k=2}^{K} \frac{1}{n_k!} \left(\frac{Q(k)}{1-Q(1)} \right)^{n_k} \right) \cdot \frac{N!}{n_1!\ (N-n_1)!}\, Q(1)^{n_1} (1-Q(1))^{N-n_1} \tag{3.22}$$

As a consequence of this property, the Multinomial distribution can be factorised as follows:

$$\begin{aligned} &\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) \\ &\quad = \mathrm{Binomial}(n_1 \mid N, Q(1)) \\ &\qquad \cdot \mathrm{Mult}\left(n_2 \ldots n_K \,\middle|\, N-n_1,\ \frac{Q(2)}{1-Q(1)} \ldots \frac{Q(K)}{1-Q(1)} \right) \end{aligned} \tag{3.23}$$

This equation can be applied recursively to decompose the multinomial distribution over $\boldsymbol{n}$ into a component-wise product of binomial distributions:

$$\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) = \prod_{k=1}^{K} \mathrm{Binomial}\left(n_k \,\middle|\, N_k,\ \frac{Q(k)}{1-\sum_{j<k} Q(j)} \right) \tag{3.24}$$

where $N_k = \left(N - \sum_{j=1}^{k} n_j\right) = \left(\sum_{j=k+1}^{K} n_j\right)$ are the remaining trials. This factorisation may also be expressed using domain restriction (defined in section 3.5.2):

$$\mathrm{Mult}(n_1 \ldots n_K \mid N,\ Q) = \prod_{k=1}^{K} \mathrm{Binomial}\left(n_k \,\middle|\, N_k,\ Q_{\setminus\{<k\}} \right) \tag{3.25}$$

Based on this insight, a basic multinomial coding method can process $\boldsymbol{n}$ sequentially and encode each n_k using a binomial code, such as the one described in section 3.4.4.

Infinite limit

It is worth pointing out that the multinomial coding method described here works even when $K = \infty$. This "infinomial distribution" is defined exactly like the multinomial distribution in (3.21), except that it has a countably infinite number of components. It is still parametrised by a finite number of trials $N \in \mathbb{N}$, but the number of possible outcomes is now infinite, i.e. the distribution Q has support over a countably infinite set.

To encode an infinomial outcome $\boldsymbol{n}$, the same algorithm can be used. Equation (3.24) applies just like in the finite case, and once N_k reaches zero the computation may safely stop since all remaining factors are then equal to 1.

3.4.7 Dirichlet-multinomial code

Just like the multinomial distribution generalises the binomial distribution to trials with K possible outcomes, the *Dirichlet-multinomial* compound distribution generalises the Beta-binomial distribution. The distribution arises from integrating out Q from a multinomial distribution, assuming that Q itself is Dirichlet distributed.

The K-dimensional Dirichlet distribution defines a probability for discrete distributions over K possible values, and is parametrized by a K-dimensional concentration vector $\boldsymbol{\alpha} = (\alpha_1 \ \ldots \ \alpha_K)$. It is therefore a distribution over distributions, defined as follows:

$$\mathrm{Dir}(Q \mid \boldsymbol{\alpha}) \ = \ \frac{\Gamma(\sum_k \alpha_k)}{\prod_k \Gamma(\alpha_k)} \cdot \prod_k Q(k)^{\alpha_k - 1} \tag{3.26}$$

The Dirichlet distribution is a multivariate generalisation of the Beta distribution, as can be seen from equations (3.17) and (3.26).

The probability mass function of the Dirichlet-multinomial can be obtained as follows:

$$\begin{aligned} \mathrm{DirMult}(\boldsymbol{n} \mid N, \ \boldsymbol{\alpha}) \ &= \ \int \mathrm{Mult}(\boldsymbol{n} \mid N, Q) \cdot \mathrm{Dir}(Q \mid \boldsymbol{\alpha}) \, dQ && (3.27) \\ &= \frac{N!}{\prod_k n_k!} \cdot \frac{\Gamma(A)}{\prod_k \Gamma(\alpha_k)} \cdot \int \prod_k Q(k)^{n_k + \alpha_k - 1} \, dQ && (3.28) \\ &= \frac{N! \ \Gamma(A)}{\Gamma(N + A)} \cdot \prod_k \frac{\Gamma(n_k + \alpha_k)}{n_k! \ \Gamma(\alpha_k)}, && (3.29) \end{aligned}$$

where $A = (\sum_k \alpha_k)$, and the components of $\boldsymbol{n}$ are non-negative integers that sum to N.

Just like with the multinomial distribution, a decomposition property can be used to split

the Dirichlet-multinomial distribution into a product:

$$\begin{aligned}&\mathrm{DirMult}(n_1 \,...\, n_K \mid (\alpha_1 \,...\, \alpha_K)\,, N)\\ &\quad= \mathrm{BetaBin}(n_1 \mid N,\ \alpha_1,\ A{-}\alpha_1)\\ &\quad\cdot\ \mathrm{DirMult}(n_2 \,...\, n_K \mid \alpha_2 \,...\, \alpha_K,\ N{-}n_1)\end{aligned} \tag{3.30}$$

This equation can be applied recursively to decompose the Dirichlet-multinomial distribution over $\boldsymbol{n}$ into a component-wise product of Beta-binomial distributions:

$$\mathrm{DirMult}(n_1 \,...\, n_K \mid \alpha_1 \,...\, \alpha_K, N) = \prod_{k=1}^{K} \mathrm{BetaBin}\left(n_k \,\middle|\, \left(\alpha_k, \sum_{j=k+1}^{K} \alpha_j\right), N - \sum_{j=1}^{k} n_j\right) \tag{3.31}$$

One can therefore compress $\boldsymbol{n}$ by sequentially encoding each component n_k using the Beta-binomial code from section 3.4.5.

Multinomial and Dirichlet-multinomial codes are used in chapter 5 for compressing unordered structures, such as multisets.

3.4.8 Codes for infinite discrete distributions

This section outlines approaches for using arithmetic coding with distributions over countably infinite sets. Such distributions include e.g. the Poisson, geometric, and negative binomial distributions. The primary difficulty of interfacing an infinite discrete distribution to a finite-precision coder is the handling of the infinite set of elements.

Without loss of generality, suppose we want to encode natural numbers $n \in \mathbb{N}$ with some known distribution P. Using the natural ordering, each element n could conceptually be mapped to $[P_\Sigma(n), P^+_\Sigma(n))$ in the unit interval. But these regions cannot be discretized into any finite range of integers, as would be required for use with a finite-precision arithmetic coder.

One solution is to factorise P in such a way that each factor involves a choice between a finite number of regions: in such a case the number of factors is generally unbounded and depends on n.

There are several ways in which such a factorisation can be made. For example, each n can be represented as a sequence of binary decisions of the form 'Is it k?' for all k from 0 to n:

$$P(n) = \prod_{k=0}^{n} \mathrm{Bernoulli}(\mathbb{1}[k{=}n] \mid \theta_k), \quad \text{where} \quad \theta_k = \frac{P(k)}{P(\{0, ..., k{-}1\})}. \tag{3.32}$$

This scheme could be viewed as a generalisation of the unary code for integers from section 2.2.1: instead of using Bernoulli choices of bias $\frac{1}{2}$, it uses Bernoulli choices of biases θ_k as shown in equation (3.32).

This method encodes each natural number n with an expected number of $\log_2 P(n)^{-1}$ bits, the Shannon information content of n under P. However, the process may be computationally inefficient for many choices of P, as the number of coding steps grows linearly with n. If the distribution P has heavy tails, or distant high mass elements, this could introduce long waits. This inefficient behaviour can be reduced by arranging the elements in descending order of probability mass, decreasing the average number of coding steps per element; but ultimately, the fact that the sequence of coding steps has a unary structure can be a computationally limiting factor for many choices of P.

Of course, many other factorisations are possible whose suitability depends on P. It should be noted that a fixed-precision arithmetic coder cannot faithfully encode choices below a probability mass of 2^{-R}, where R is the coder's current coding range. The factors must therefore be chosen such that its probabilities lie sufficiently above the limiting threshold.

Another approach to this problem might be to ignore elements whose probability is below a certain threshold; and treat the occurrence of such low-probability events as an error condition.

3.5 Combining distributions

Probabilistic models can be constructed by combining simpler distributions. This section describes coding techniques for some common ways in which probability distributions can be combined.

3.5.1 Sequential coding

Recall from section 3.2 that the sequential arithmetic encoding of symbols $x_1 \ldots x_N$ with associated probability distributions $P_1 \ldots P_N$ produces a final region whose implied probability mass corresponds to the product of the individual symbol probabilities:

$$P(x_1 \ldots x_N) \;=\; \prod_{n=1}^{N} P_n(x_n). \tag{3.33}$$

If we want to arithmetically encode a multivariate vector $(x_1 \ldots x_N)$ for which only the joint distribution is known, we may have to factorise the distribution into marginal and conditional distributions first. For example, a distribution over two variables can be factorised in two ways:

$$\Pr(x, y) \;=\; \Pr(x) \cdot \Pr(y \mid x) \;=\; \Pr(y) \cdot \Pr(x \mid y) \tag{3.34}$$

To encode values x and y according to their joint distribution, one could first encode x with the marginal distribution $\Pr(x)$ and then encode y with the conditional distribution $\Pr(y \mid x)$.

This construction preserves optimality and always works, but requires a coding method for the marginal and conditional distributions.

It can help to choose the marginal: sometimes $\Pr(x \mid y)$ is easier to encode than $\Pr(y \mid x)$. If conditional distributions are difficult to compute, but the marginals are not, it might be tempting to use *only* the marginals: i.e. coding x with $\Pr(x)$, then y with $\Pr(y)$. While using such an encoding still allows (x, y) to be reconstructed, the wrongly assumed independence of x and y may incur a bandwidth cost.[5]

More generally, factors in a probabilistic model turn into sequential coding steps. The order of the steps must be such that each step depends only on information that has already been encoded, so that the decoder can reconstruct each conditional distribution in the same order. Which marginals or conditionals are chosen does not matter, and can be chosen for computational convenience.

3.5.2 Exclusion coding

One particular form of conditional distribution is a *value exclusion* or *domain restriction*, where a subset $\mathcal{R}$ of values is excluded from the choice:

$$P(x \mid x \notin \mathcal{R}) \;=\; \frac{P(x \notin \mathcal{R} \mid x) \cdot P(x)}{P(x \notin \mathcal{R})} \;=\; \frac{\mathbb{1}[x \notin \mathcal{R}] \cdot P(x)}{1 - P(\mathcal{R})} \tag{3.35}$$

This conditional distribution is well defined as long as $P(\mathcal{R}) < 1$. We'll abbreviate this conditional distribution $P_{\setminus\mathcal{R}}$:

$$P_{\setminus\mathcal{R}}(x) \;=\; \begin{cases} 0 & x \in \mathcal{R} \\ \dfrac{P(x)}{1 - P(\mathcal{R})} & \text{otherwise} \end{cases} \tag{3.36}$$

Sampling from $P_{\setminus\mathcal{R}}$ is the same as sampling from P and rejecting values $x \in \mathcal{R}$. (Of course there may be more efficient ways to sample from $P_{\setminus\mathcal{R}}$ than rejection sampling.)

Encoding or decoding values x under a conditional distribution of the above kind is called *exclusion coding*. For data compression algorithms, exclusion coding can be a useful tool for several reasons:

1. It provides a simple way to incorporate knowledge into an existing model that certain values cannot occur, saving otherwise wasted information.

2. It preserves the relative proportions of probability mass of non-excluded elements, and

[5] For example, consider the distribution $\Pr(x, y) = \mathbb{1}[x = y]$ where x and y take values in $\{0, 1\}$. The marginal distributions are $\Pr(x) = \Pr(y) = \frac{1}{2}$. Encoding a tuple (x, y) with $\Pr(x) \cdot \Pr(y \mid x)$ costs 1 bit, whereas coding x and y independently with $\Pr(x)$ and $\Pr(y)$ costs 2 bits.

can be used as a mathematical tool to encode components of multivariate distributions. Such use occurred e.g. in section 3.4.6, equation (3.25).

3. Domain restrictions can be used to replace a mixture model with a computationally favourable alternative, by enforcing the domains of the mixture components to be disjoint. Separating the mixture components in this way lowers the cost of encoding and decoding.

Value exclusions are used implicitly in some data compression algorithms, for example in the PPM algorithm by Cleary and Witten (1984a) and many of its variants. PPM algorithms and their corresponding probabilistic models are covered in detail in chapter 6.

3.5.3 Mixture models

A mixture model combines a finite or countable number of distributions according to a mixing distribution. For example, consider a family of distributions D_1 ... D_K over some space $\mathcal{X}$, and a discrete distribution Θ over the index set $\{1 \dots K\}$, to define a *mixture distribution* M:

$$M(x) \;=\; \Pr(x \mid \Theta, D_1 \;...\; D_K) \;=\; \sum_{k=1}^{K} \Theta(k) \cdot D_k(x) \tag{3.37}$$

The distribution Θ is called the *mixing distribution*. Sampling from M is easy: first draw an index k from Θ, then draw a value x from the kth component distribution D_k (and throw away k). To obtain the probability mass $M(x)$ of an outcome x, the contributions of each component distribution D_k must be summed (unless the sum has a closed form).

To arithmetically encode a value x with a mixture distribution M, discrete region boundaries must be computed, for example by using the mixture's cumulative probability $M_\Sigma(x)$.

If the mixing distribution Θ changes more frequently than the component distributions D_k, it may help to compute $M_\Sigma(x)$ as follows:

$$M_\Sigma(x) \;=\; \sum_{k=1}^{K} D_{k\Sigma}(x) \tag{3.38}$$

where the cumulative component distributions $D_{k\Sigma}$ can be cached to speed up the computation.

Chapter 4

Adaptive compression

The previous chapter reviewed how arithmetic coding can be used for compressing sequences of symbols whose distributions are known. However, in most practical compression applications, the distributions aren't known in advance. In such cases, adaptive techniques are useful: these methods compress sequences while simultaneously learning their symbol distributions. Building on techniques from the previous chapters, this chapter presents several basic adaptive compression methods that find use in later algorithms. Finally, the chapter motivates the work on context-sensitive models and structural compression.

4.1 Learning a distribution

4.1.1 Introduction

Suppose we'd like to compress a sequence of symbols $x_1 \ldots x_N$ from a finite symbol alphabet where each x_n is independent and identically distributed (*iid*) according to some *unknown* distribution D, and the goal is to minimise the expected length of the compressed output. Because D is unknown, there is no other option than using different distributions Q_n for compressing the symbols: the closer these guessed distributions Q_n are to the unknown D, the better will be the compression effectiveness.[1]

Although the latent distribution D is initially unknown, every symbol x_n of the sequence conveys some information about it: adaptive compression methods exploit this information to construct estimates Q_n of D from the previous symbols $x_1 \ldots x_n$, and uses Q_n to compress the next symbol x_{n+1}. If done correctly, the estimated distributions Q_n will *adapt* to the

[1] The expected penalty for encoding a D-distributed symbol with the wrong distribution Q is $\mathrm{KL}(D \,||\, Q)$ nats (Cover and Thomas, 1991, Theorem 5.4.3), where KL denotes the relative entropy, or Kullback–Leibler divergence (Kullback and Leibler, 1951). 1 nat equals $(\ln 2)^{-1}$ bits.

input data and converge to the true distribution D. One of the benefits of using arithmetic coding is that a different distribution can be used for every symbol in the sequence, at little additional cost.

4.1.2 Motivation

Why would we need to encode symbols from an unknown distribution? As a motivational example, consider a sequence of English text. An empirically determined distribution over letters in English was published by Norvig (2013), working from the Google Books N-gram corpus that includes millions of English language books (Michel et al., 2011). This distribution is shown in Figure 4.1.

However, even if the true global symbol distribution were known, the symbol distribution of any particular document is likely to differ from the global average. For example the distribution of symbols in this thesis, shown in Figure 4.2, is visibly different from Norvig's average.

Yet another motivation is that many sequences may be written in a language other than English, or may not be human language data at all. In the absence of prior knowledge, it is both elegant and useful to learn the distribution from the data itself.

Note. The adaptive compression methods presented in this chapter focus on learning a single symbol distribution for each input sequence. Chapter 6 discusses generalisations that learn not just one global symbol distribution, but many context-dependent distributions; these distributions differ massively from the mean shown in Figure 4.1, and are likely to be more distinctive to each input sequence.

4.2 Histogram methods

4.2.1 A simple exchangeable method for adaptive compression

Adaptive compression techniques encode each symbol in a sequence with a probability distribution derived from the previous symbols in the sequence. Commonly, the probability that the N+1st symbol x_{N+1} equals a given value x is defined as a function of the number of times x was observed in the preceding sequence x_1 ... x_N.

Let's assume that the sequence x_1 ... x_N was generated by making independent draws from an unknown symbol distribution D over a symbol alphabet $\mathcal{X}$. One way of adaptively compressing such a sequence is to encode each symbol with a distribution that approximates D

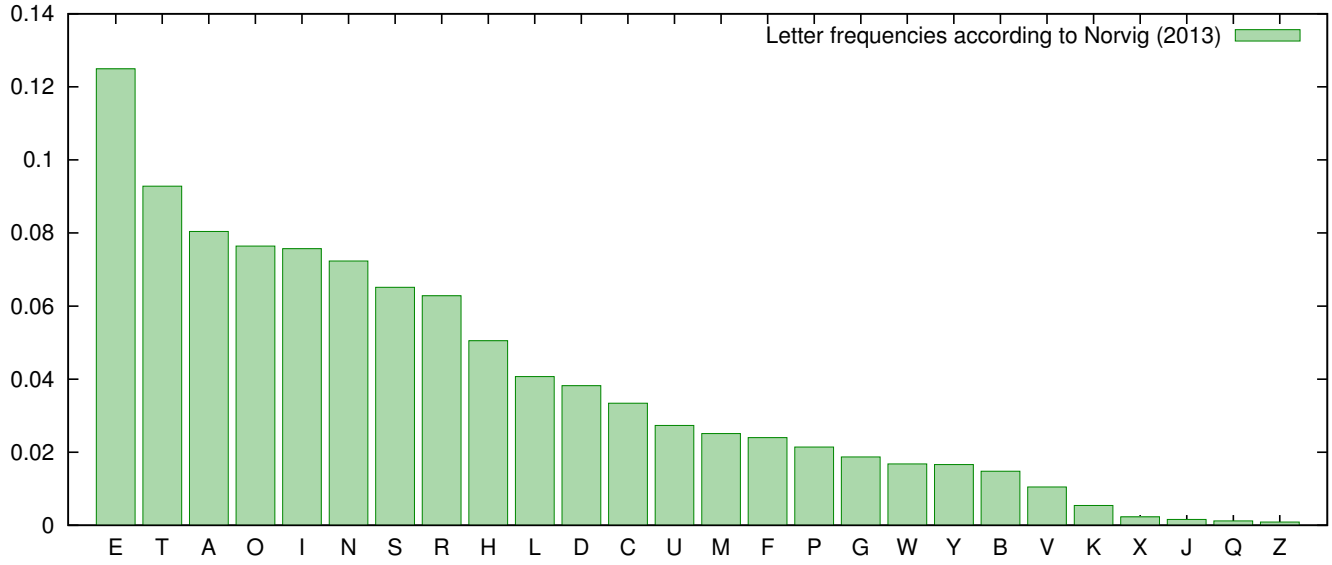


Figure 4.1: Symbol frequencies of English text, as measured by Norvig (2013) using the Google Books N-gram corpus (Michel et al., 2011). Uppercase and lowercase letters have been merged and all other symbols removed.

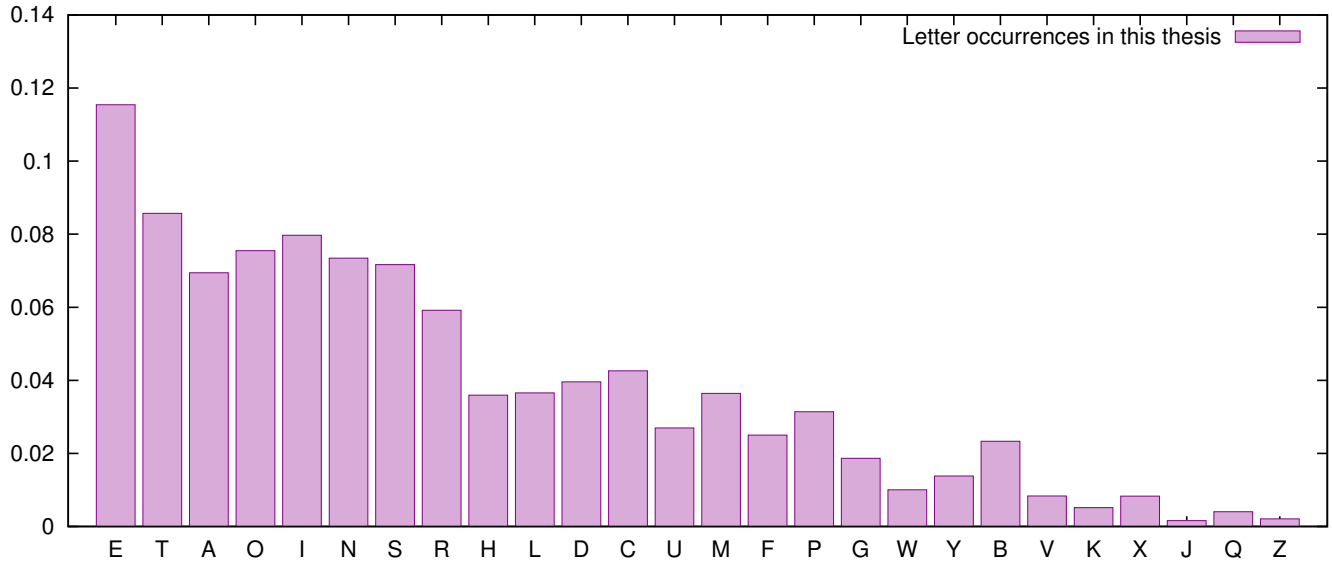


Figure 4.2: Symbol frequencies in this thesis. For ease of comparison, the symbols are presented in the same order as in Figure 4.1.

from the occurrences of the preceding symbols, for example as follows:

$$\Pr(x_{N+1} = x \mid x_1 \ldots x_N) = \frac{\#[x_n = x]_{n=1}^{N} + 1}{N + T} \tag{4.1}$$

Here, $T = |\mathcal{X}|$ is the total number of symbols in the alphabet, and $\#[x_n = x]_{n=1}^{N}$ denotes the number of times symbol x appears in sequence $x_1 \ldots x_N$, satisfying:

$$\#[x_n = x]_{n=1}^{N} = \sum_{n=1}^{N} \mathbb{1}[x_n = x] \qquad \text{and} \qquad \sum_{x \in \mathcal{X}} \#[x_n = x]_{n=1}^{N} = N. \tag{4.2}$$

Noting that arithmetic coding makes it easy to use a different distribution for each symbol, equation (4.1) can be interfaced to an arithmetic coder in straightforward fashion. Computing the required cumulative symbol distributions may be slightly more elaborate than for static compression, as the coding distribution changes for every symbol; to mitigate this problem, designated dynamic data structures exist that provide access to cumulative symbol counts at much reduced computational cost (Fenwick, 1993, 1995; Moffat, 1999).

Exchangeability. The information content of a sequence whose elements are independent and identically distributed (*iid*) does not depend on the order of the symbols. The compressed output length produced by an appropriately derived compression method should therefore also be order-invariant. This property holds for the adaptive compression scheme described above. Sampling an infinite sequence from the stochastic process defined in (4.1) produces a random symbol distribution D, where $D \sim \text{Dirichlet}\big(\boldsymbol{\alpha} = (\underbrace{1, 1, \ldots, 1}_{|\mathcal{X}|})\big)$.

More generally, the output length of an adaptive compressor is order-invariant if the compressor's underlying stochastic process produces infinitely exchangeable sequences. The relationship between exchangeable sequences and conditional independence given some latent variable is characterised in the theorem of de Finetti (1931).

Many existing adaptive compression methods do not satisfy this exchangeability property, for example the move-to-front encoder described in section 2.4.1, and most smoothing methods based on backing-off (described in chapter 6). There are many ways of designing adaptive compression algorithms, and each such method may embed different assumptions about the unknown symbol distribution. But any method whose compression effectiveness depends on the order of the symbols in the sequence necessarily violates the assumption that the symbols are independent and identically distributed according to an unknown symbol distribution. (Of course, making an *iid* assumption is not reasonable for many kinds of input sequence, including human text.)

The adaptive compression scheme defined in equation (4.1) is a simple instance of a histogram-building method, and a special case of the urn scheme by Blackwell and MacQueen (1973). Several other popular (and mostly non-exchangeable) histogram-building methods exist; an

extensive survey can be found in the report by Chen and Goodman (1998). Many of these methods have been applied to data compression.

4.2.2 Dirichlet processes

The histogram-building method described in the previous section gives each symbol an initial count of 1. These initial counts constitute one way of ensuring that every symbol has a non-zero probability of occurrence, and that the distribution over symbols starts out uniform.

This basic approach can be generalised by allowing an arbitrary base distribution G, and introducing a strength parameter α that controls how quickly the base distribution is influenced by empirical symbol counts. A sequential form of this generalised approach can be written as follows:

$$\Pr(x_{N+1} = x \mid x_1 \ldots x_N) = \frac{\#[x_n = x]_{n=1}^{N}}{N + \alpha} + \frac{\alpha}{N + \alpha} G(x) \tag{4.3}$$

where α is the strength parameter, and G is the base distribution. A common choice for G might be a uniform distribution over the symbol space $\mathcal{X}$, but any discrete distribution can be used (including distributions with infinite support). The parameter α controls the blending of the empirical distribution with the base distribution G: high values of α give more weight to G, whereas low values of α let the empirical distribution dominate.

Theoretical background. The stochastic process defined by sequential application of the conditional distribution (4.3) corresponds to the urn schemes of Blackwell and MacQueen (1973), Hoppe (1984), and the Chinese restaurant process of Aldous (1985).[2] Sequences produced from such an urn are infinitely exchangeable, and the symbol distribution obtained from (4.3) as $N \to \infty$ is called a *Dirichlet process* (Ferguson, 1973). An introduction to Dirichlet processes can be found in e.g. Walker et al. (1999), Ghosh and Ramamoorthi (2002), and Teh (2010).

It is possible to construct hierarchical Dirichlet processes by letting G itself be a symbol distribution distributed according to a Dirichlet process prior. Hierarchical Dirichlet processes and mechanisms for learning the latent base distributions are described by Teh et al. (2006).

4.2.3 Power-law variants

The urn scheme described in section 4.2.2 can be generalised to variants that induce power-law behaviour. Two such variants are presented below, each involving the addition of a *discount parameter* β.

[2] The original form of the Chinese restaurant process (CRP) describes partition structures (sequences of cluster indices $c_1 \ldots c_N$, or "table assignments"), rather than sequences of elements $x_1 \ldots x_N$. Equation (4.3) can be obtained from Aldous' CRP by sampling an element $x_c \in \mathcal{X}$ for each cluster index c, and mapping the sequence of indices $c_1 \ldots c_N$ to the sequence of elements $x_{z_1} \ldots x_{z_N}$.

The first variant is an instance of 'interpolated Kneser–Ney smoothing' as defined by Chen and Goodman (1998). It may also be called a two-parameter Chinese restaurant process in which no two tables can serve the same dish. The sequential construction of a sequence with such a distribution extends equation (4.3) as follows:

$$\Pr(x_{N+1} = x \mid x_1 \ldots x_N) = \frac{n_x - \beta}{N + \alpha} \mathbb{1}[n_x > 0] + \frac{\alpha + \beta U_N}{N + \alpha} G(x), \tag{4.4}$$

where U_N is the number of unique symbols in $x_1 \ldots x_N$, and n_x abbreviates $\#[x_n = x]_{n=1}^{N}$. The factor $\mathbb{1}[n_x > 0]$ ensures that the discount β is only applied to symbols that have occurred at least once. Construction (4.5) is used, for example, in the hierarchical context compressors BPPM and UKN-Deplump that are described in sections 6.5 and 6.6.4.

The second variant (which is closely related to the first) is the standard two-parameter Chinese restaurant process, which is a sequential construction for the Pitman–Yor process:

$$\Pr(x_{N+1} = x \mid x_1 \ldots x_N, \boldsymbol{t}) = \frac{n_x - t_x\beta}{N + \alpha} + \frac{\alpha + \beta T}{N + \alpha} G(x) \tag{4.5}$$

When viewed as a generative sampling procedure, n_x counts how many (out of N) times x was generated in the preceding sequence $x_1 \ldots x_N$, and t_x counts how often (out of those n_x times) x was generated from G as opposed to from the first component. The quantity T is the sum of the t_x for all $x \in \mathcal{X}$. Note that unlike the $\boldsymbol{n}$, the latent $\boldsymbol{t}$ cannot be deterministically computed from $x_1 \ldots x_N$, as their values depend on external randomness. However, the $\boldsymbol{t}$ can be marginalised out using stochastic inference procedures; methods for doing this are described by e.g. Teh (2006b).

The Pitman–Yor process is described by Perman (1990); Perman, Pitman and Yor (1992) and Pitman and Yor (1995); and given its name by Ishwaran and James (2001).

The generalisation to hierarchical Pitman–Yor processes eventually led to the Sequence Memoizer model by Wood et al. (2009, 2011), which has been applied to compression by Gasthaus et al. (2010), and Bartlett and Wood (2011). The Sequence Memoizer and its relation to the PPM algorithm are discussed in section 6.6.

4.2.4 Non-exchangeable histogram learners

Many other learning mechanisms exist that construct a symbol histogram (or any collection of symbol occurrence counts) to approximate the true symbol distribution. Many such methods can be found in the report by Chen and Goodman (1998). Some of these also have discount parameters, and some also exhibit power-law behaviour. Examples of such constructions include some of the escape methods of the PPM algorithm (Cleary and Witten, 1984a), which are described in section 6.3. Unlike the Pitman–Yor process, however, these constructions do

not generally define exchangeable stochastic processes.

One may argue that exchangeability isn't always important in practice, as few sequences consist of symbols that are independent and identically distributed. However, models that produce exchangeable sequences are much easier to combine and reason about than their non-exchangeable counterparts, facilitating flexible engineering with more predictable results.

4.3 Other adaptive models

The adaptive models from section 4.2 can learn an underlying symbol distribution P of any sequence whose symbols are independent and identically distributed according to P. These adaptive models are similar in that their mechanism of learning involves building an explicit histogram of symbol occurrences. But there are other ways of constructing adaptive models, and one such alternative is the Pólya tree method discussed below.

Many of the algorithms from chapter 2 are also adaptive, and their learning mechanisms may involve the construction of something other than a histogram, such as a ranked list (e.g. by move-to-front encoders) or a dictionary. Figure 4.3 compares the compression effectiveness of various adaptive models on a pseudo-randomly generated sequence of independent and identically distributed symbols (from a random symbol distribution).

4.3.1 A Pólya tree symbol compressor

The methods from section 4.2 assign probability mass to symbols based on a histogram of previous symbol occurrences. The histogram can be represented as a collection of integer counts $\boldsymbol{n}$, one for each symbol x in the alphabet $\mathcal{X}$. An alternative method is now presented that does not build an explicit representation of a histogram, but still defines an exchangeable process.

The Pólya tree method uses a balanced binary search tree whose leaf nodes contain the symbols of the alphabet $\mathcal{X}$, such that each symbol $x \in \mathcal{X}$ can be identified by a sequence of (at most $\lceil \log_2 |\mathcal{X}| \rceil$) binary branching decisions from the root of the tree. The tree has $K = |\mathcal{X}| - 1$ internal nodes (labelled with integers $k = 1, \ldots, K-1$), each containing a value θ_k that represents the probability of choosing between its two children.

The probability of a given symbol x is then defined as the product of the probabilities of the branching decisions taken to reach x from the root:

$$P(x) = \prod_{k \in \mathsf{PATH}(x)} \text{Bernoulli}(b_k \mid \theta_k) \tag{4.6}$$

where $\mathsf{PATH}(x)$ denotes the set of nodes k that lie on the path from the root to x, and

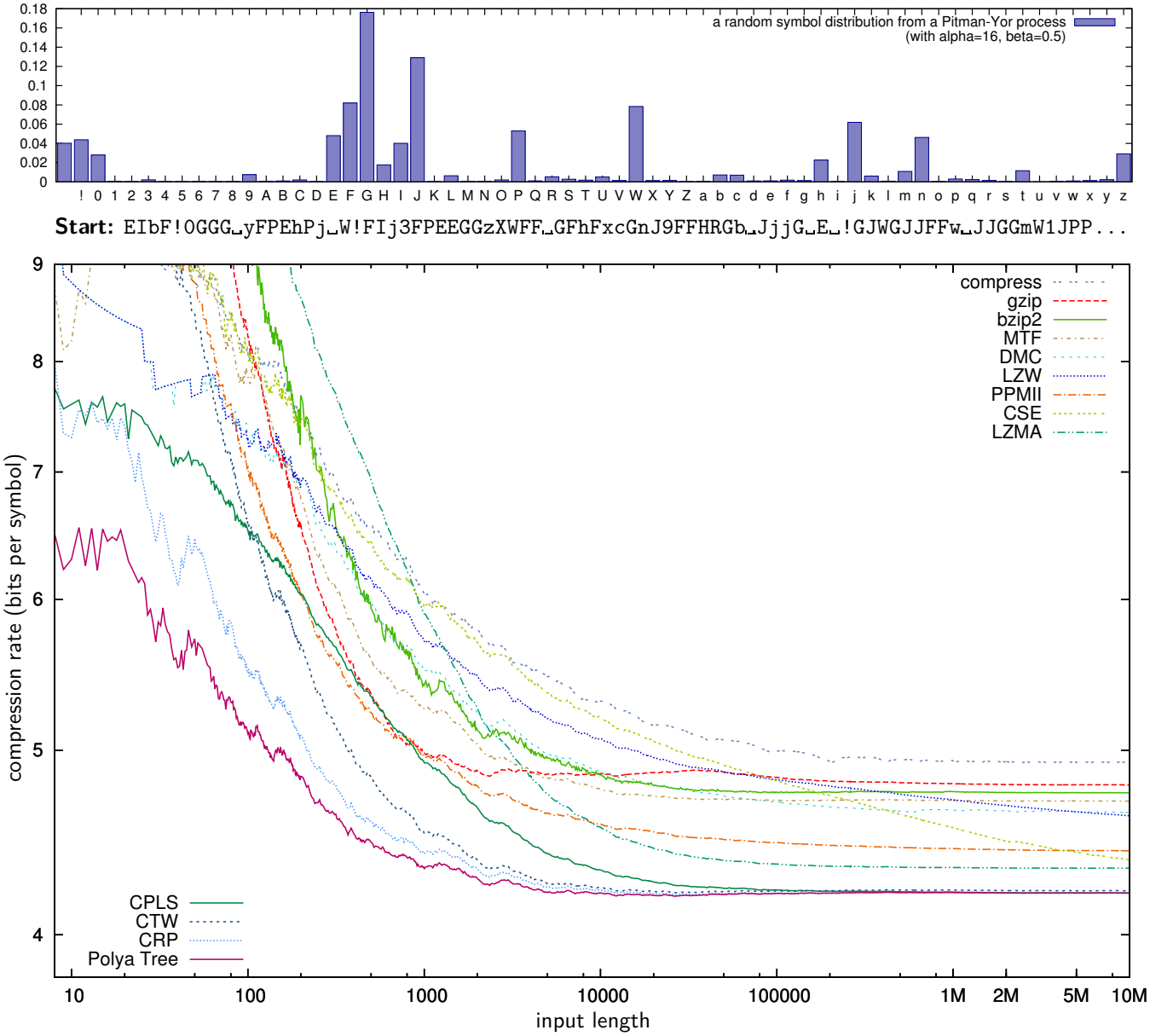


Figure 4.3: This plot shows the compression effectiveness of various adaptive compression methods on a sequence of pseudo-random symbols, drawn *iid* from a random symbol distribution P. The start of the sequence is shown between the two graphs. P was sampled from a Pitman–Yor process with concentration $\alpha = 16$ and discount $\beta = \frac{1}{2}$, and with a uniform base distribution over a set of 64 symbols $\{\text{␣}, \texttt{!}, \texttt{0–9}, \texttt{A–Z}, \texttt{a–z}\}$. A histogram of P is shown in the graph at the top.

Some adaptive compression methods learn the underlying symbol distribution more rapidly than others, and converge to a better compression rate more quickly.

The programs compress, gzip, bzip2, and LZMA are described in chapter 2. MTF is a move-to-front encoder (as described in 2.4.1), followed by an adaptive model that learns the integer indices (sec. 4.2.1). The line labelled LZW is my own implementation of the LZW algorithm, using unlimited memory and arithmetic coding. CPLS is an implementation of the histogram learner from section 4.2.1, using a cumulative probability lookup structure in the spirit of Fenwick (1995). CRP is a histogram learner using a Pitman–Yor process as described in section 4.2.3. Polya Tree is an implementation of the Pólya tree symbol learner described in section 4.3.1. An index of all algorithms can be found in appendix A.

$b_k \in \{0, 1\}$ identifies the branching decision at node k for reaching x. This construction is called a discrete domain Pólya tree (Ferguson, 1974; Lavine, 1992; Mauldin et al., 1992). Any discrete distribution P over $\mathcal{X}$ can be expressed by choosing appropriate node biases θ_k in (4.6). An example is shown in Figure 4.4.

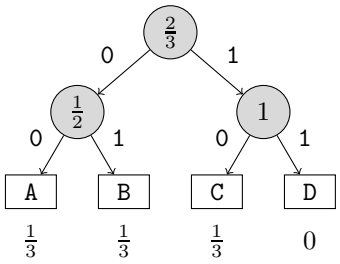


Figure 4.4: A Pólya tree representing a distribution over the alphabet $\mathcal{X} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ as a sequence of weighted binary choices. The distribution described by this tree has $\Pr(\mathtt{A}) = \Pr(\mathtt{B}) = \Pr(\mathtt{C}) = \frac{1}{3}$ and $\Pr(\mathtt{D}) = 0$. The values shown in the shaded nodes represent the probability of following the node's 0-branch.

With a Pólya tree representation of a distribution P, a P-distributed symbol x can be arithmetically encoded as the sequence of branching decisions that identifies x, where each b_k is encoded with Bernoulli code of bias θ_k. This method requires knowing the θ_k (and hence P).

Consider now the case that P is unknown. The Pólya tree compressor can be made adaptive by adding a mechanism that learns the node biases from the data. A Bayesian way of deriving such a learning mechanism is to place priors on the node biases, for example $\theta_k \sim \text{Beta}(\alpha, \beta)$, and use Bayes' rule to obtain a posterior distribution over θ_k given the branching decisions (at node k) of all previous symbols that traversed node k.

Due to the conjugacy of the Beta and Bernoulli distributions, the posterior distributions over the θ_k are still Beta distributions. Furthermore, the biases θ_k can be integrated out to give a Beta-Bernoulli compound distribution of the form:

$$\text{BetaBernoulli}(b_k \mid \alpha, \beta) = \int \text{Bernoulli}(b_k \mid \theta_k) \cdot \text{Beta}(\theta_k \mid \alpha, \beta)\, d\theta_k \tag{4.7}$$

$$= \left(\frac{\alpha}{\alpha+\beta}\right)^{1-b_k} \left(\frac{\beta}{\alpha+\beta}\right)^{b_k} \tag{4.8}$$

$$= \text{Bernoulli}\left(b_k \,\middle|\, \frac{\alpha}{\alpha+\beta}\right) \tag{4.9}$$

A definition of the Beta distribution can be found in equation (3.17). Given a local history of M branching decisions $b_k^{(1)} \ldots b_k^{(M)}$ at node k, the predictive probability of the next branching decision $b_k^{(M+1)}$ is given by:

$$\text{BetaBernoulli}\left(b_k^{(M+1)} \,\middle|\, \alpha, \beta,\ b_k^{(1)} \ldots b_k^{(M)}\right) = \text{Bernoulli}\left(b_k^{(M+1)} \,\middle|\, \frac{\alpha + m_k^{(0)}}{\alpha + \beta + M}\right), \tag{4.10}$$

$$\text{where}\quad m_k^{(0)} \stackrel{\text{def}}{=} \#\left[b_k^{(m)} = 0\right]_{m=1}^{M} \qquad \left(\text{and } m_k^{(1)} \stackrel{\text{def}}{=} M - m_k^{(0)}\right) \tag{4.11}$$

The quantity $m_k^{(0)}$ counts how many of the M node traversal followed the `0`-branch, and $m_k^{(1)}$ counts those that followed the `1`-branch.

To implement an adaptive Pólya tree compressor, it is sufficient to store $m_k^{(0)}$ and $m_k^{(1)}$ at each node k. Compression results of such an implementation (with $\alpha = \beta = \frac{1}{2}$ for all nodes) are included in the comparison of Figure 4.3. The asymptotic behaviour of adaptive Pólya tree compressors is identical to that of the adaptive histogram building methods. However, the Pólya tree model may learn certain probability distributions more quickly, for example distributions whose probability mass is similar for groups of symbols that are located closely together in the tree. Such distributions commonly arise in e.g. multilingual text documents in Unicode, where the occurrence of symbols of one script (such as Latin, Greek, Cyrillic, Arabic, traditional Chinese, etc.) often means that more symbols of the same script are likely to follow. Adaptive Pólya tree compressors can be constructed over the Unicode alphabet (which has designated blocks of adjacent code points for the symbols of different scripts), and can be made hierarchical in similar fashion as the more traditional histogram-building models can. However, this idea is not pursued further in this thesis, and the remainder of this chapter focuses on the more widely known histogram-building methods.

4.4 Online versus header-payload compression

Sections 4.2 and 4.3 introduced adaptive compression methods for sequences of independent and identically distributed symbols, that gradually learn the symbol distribution *online* while the sequence is being compressed. This section compares such adaptive online methods with *header-payload* methods, in which the optimal symbol distribution is computed and communicated to the receiver first, and then the sequence is sent in a way that exploits that optimal distribution.

Examples of header-payload methods include compressors that use Huffman coding and transmit the optimal Huffman tree before sending the data. Such a technique is used in e.g. the DEFLATE algorithm (see section 2.3.2), where Huffman coding is used to compress the output of the LZ77 stage, and the Huffman tree is transmitted in the form of a compressed set of code word lengths.

Some people might say that either approach has merits and drawbacks; that learning the distribution gradually means that the first symbols are encoded ineffectively, and that the header-payload method is suboptimal because of redundancy between the header and the payload. But both views are incorrect: a correctly implemented online method doesn't transmit the early symbols ineffectively – it conveys them using exactly the required bandwidth given the receiver's uncertainty. Secondly, as will be shown in section 4.4.2, the header-payload method can be done in a way that achieves the optimal compressed length, too.

How can these strategies be implemented correctly? How should one optimally encode a known symbol distribution? And which approach is better? Answers to these questions yield useful insights for optimal encoding of multisets, lengths and orderings.

For concreteness, consider a sequence of N characters $x_1 \ldots x_N$ drawn *iid* from a distribution P over a finite symbol alphabet $\mathcal{X}$. The probability of this sequence, given N and P, is:

$$\Pr(x_1 \ldots x_N \mid P, N) = \prod_{n=1}^{N} P(x_n) \tag{4.12}$$

The aim of any adaptive compressor is to learn the unknown distribution P and use this knowledge to compress the sequence $x_1 \ldots x_N$. I will present two alternative approaches for solving this problem, each derived from first principles.

4.4.1 Online compression

The approach of *online compression* is to encode the sequence one symbol at a time, gradually learning the original distribution P. Encoding one symbol at a time corresponds to factorising

the joint probability of the sequence as follows:

$$\Pr(x_1 \ldots x_N \mid N) = \prod_{n=1}^{N} \Pr(x_n \mid x_1 \ldots x_{n-1}) \tag{4.13}$$

The distribution over each x_n embodies our belief over P, taking into account the symbols of the preceding sequence:

$$\Pr(x_n \mid x_1 \ldots x_{n-1}) = \int \Pr(x_n \mid P) \cdot \Pr(P \mid x_1 \ldots x_{n-1}) \; dP \tag{4.14}$$

The gradual learning of P can be done through Bayesian inference, using a suitable prior for P. For example, a Dirichlet prior is a convenient choice:

$$\Pr(P \mid \boldsymbol{\alpha}) = \frac{\Gamma(A)}{\prod_{x \in \mathcal{X}} \Gamma(\alpha_x)} \cdot \prod_{x \in \mathcal{X}} P(x)^{(\alpha_x - 1)} \tag{4.15}$$

Here $\boldsymbol{\alpha}$ is a $|\mathcal{X}|$-length parameter vector, and A is the sum of all α_x. The posterior distribution of P, given observations $x_1 \ldots x_{n-1}$, is:

$$\Pr(P \mid \boldsymbol{\alpha},\ x_1 \ldots x_{n-1}) = \frac{\Gamma(A + (n-1))}{\prod_{x \in \mathcal{X}} \Gamma\Big(\alpha_x + \#[x_k = x]_{k=1}^{n-1}\Big)} \cdot \prod_{x \in \mathcal{X}} P(x)^{(\alpha_x - 1 + \#[x_k = x]_{k=1}^{n-1})} \tag{4.16}$$

Integrating out P, we recover the familiar looking closed-form solution[3] for the conditional distribution over x_n:

$$\Pr(x_n = x \mid \boldsymbol{\alpha},\ x_1 \ldots x_{n-1}) \;=\; \frac{\alpha_x + \#[x_k = x]_{k=1}^{n-1}}{A + n - 1} \tag{4.17}$$

This result is useful, because it gives a fast, incremental way of computing the predictive distribution over the next symbol. To get the probability mass of symbol x_n, it is sufficient to count the number of times it occurred before, add a fixed value α_x, and do a division. The counts can be stored in an array of size $|\mathcal{X}|$, which is easy to query and update. Indeed, algorithms operating according to equation (4.17) have existed for a long time.[4]

To conclude, let's substitute equation (4.17) back into equation (4.13), to obtain the joint distribution over the entire sequence, as induced by the online compression method:

$$\Pr(x_1 \ldots x_N \mid \boldsymbol{\alpha}, N) = \prod_{n=1}^{N} \frac{\alpha_{x_n} + \#[x_k = x]_{k=1}^{n-1}}{A + n - 1} \tag{4.18}$$

This distribution is optimal in the sense that the coding of each symbol makes best use of all available knowledge from the preceding sequence.

[3] The predictive distribution (4.17) generalises (4.1), and is generalised by (4.3) and (4.5).

[4] Parameters α_x are sometimes set to 1 for all $x \in \mathcal{X}$: this special case is also known as the *rule of succession* or the *Laplace estimator* (Laplace, 1814).

4.4.2 Header-payload compression

An alternative method for encoding a sequence of independent and identically distributed symbols is to transmit (an approximation of) the distribution P first, and then use it to optimally code the source symbol sequence. Encoding P first clearly costs some bandwidth. This might seem wasteful at first glance, especially when the symbol sequence is short: consider for example the case $N = 1$, where only one symbol needs transmitting, or $N = 0$, encoding an empty sequence.

To make matters worse, encoding P exactly is not even possible technically, as a distribution is not generally a finite object (even when its support is a discrete finite alphabet).[5] Certainly, header-payload compression can be done badly; many header-payload compression methods waste bits by sending redundant information. But header-payload compression can be done optimally.

The key realisations are that for optimal encoding and decoding of a *finite* input sequence, a finite approximation of P is sufficient, and that the sequence can be encoded using not that fixed approximation, but an approximation that adapts appropriately as the sequence is sent. The amount of precision needed for this approximation depends on N, the sequence length.

If we know that the length of the sequence is N, we can instead, for each symbol x in the alphabet $\mathcal{X}$, count the number of times it occurs in the sequence, written $\mathcal{N}(x)$, where $\sum_{x\in\mathcal{X}} \mathcal{N}(x) = N$. Let's call the collection $\mathcal{N}$ of these counts the *symbol histogram* of the sequence. For any finite sequence of known length, the distribution of symbols is fully captured by $\mathcal{N}$.

The *header-payload compression* idea is to decompose the joint probability of the sequence as follows:

$$\Pr(x_1 \ldots x_N \mid N) = \Pr(\mathcal{N} \mid N) \cdot \Pr(x_1 \ldots x_N \mid \mathcal{N}) \tag{4.19}$$

This factorisation separates the sequence's *histogram* from its *permutation*. Due to the conditional independence of these two components, a compressor could encode the histogram $\mathcal{N}$ first (conditional on the length), followed by the symbol permutation (conditional on the histogram).

Encoding methods for each of these two tasks are derived below. First, I will show how a symbol sequence $x_1 \ldots x_N$ can be encoded optimally, assuming its symbol histogram $\mathcal{N}$ is known. Then I will describe an optimal coding scheme for the histogram $\mathcal{N}$ itself, assuming N is known.

[5] For example, consider a distribution over two symbols {A,B} with $\Pr(\mathtt{A}) = \frac{1}{\xi}$ and $\Pr(\mathtt{B}) = \frac{\xi-1}{\xi}$, where ξ is an arbitrary irrational number greater than 1. Encoding an exact representation of this distribution involves encoding an exact representation of ξ, which may require transmitting an arbitrarily large amount of information.

Optimal encoding of the symbol permutation

This section discusses how to encode the input sequence $x_1 \ldots x_N$ when its symbol counts $\mathcal{N}$ are known. If we did this naïvely using a fixed 'optimal' distribution, then information would be wasted, since each symbol tells us something about the remaining symbols.

For example, consider the case of an input sequence of 100 symbols, of which 99 are `A`, and exactly 1 is `B`. The state of this sequence is captured entirely by the position of the `B` symbol. From the point of view of a decoder that processes the sequence symbol by symbol, the moment `B` is decoded marks the point at which no further information is needed, as all remaining symbols must be `A`.

Similarly, consider a sequence of N unique symbols. The symbol x_n at the nth position in this sequence is optimally encoded as a choice between the $N - n$ remaining symbols, since those which have occurred now have zero probability mass.

In general, an arbitrary sequence $x_1 \ldots x_N$ with histogram $\mathcal{N}$ can be optimally encoded by encoding each symbol x_n sequentially, taking into account the preceding sequence:

$$\Pr(x_n = x \mid \mathcal{N}, x_1 \ldots x_{n-1}) = \frac{\mathcal{N}(x) - \#[x_k = x]_{k=1}^{n-1}}{N - n + 1} \tag{4.20}$$

This is a *strike-one-off* approach: after a symbol is encoded, its count is decremented by one, improving the prediction of the remaining symbols.

The total probability of the entire sequence then becomes:

$$\begin{aligned}
\Pr(x_1 \ldots x_N \mid \mathcal{N}) &= \prod_{n=1}^{N} \Pr(x_n \mid \mathcal{N}, x_1 \ldots x_{n-1}) && (4.21)\\
&= \prod_{n=1}^{N} \frac{\mathcal{N}(x_n) - \#[x_k = x_n]_{k=1}^{n-1}}{N - n + 1} && (4.22)\\
&= \frac{1}{N!} \prod_{n=1}^{N} \left(\mathcal{N}(x_n) - \sum_{k=1}^{n-1} \mathbb{1}[x_k = x_n] \right) && (4.23)
\end{aligned}$$

The product can be rewritten to range over the symbol alphabet rather than the sequence positions, yielding the following concise form:

$$\Pr(x_1 \ldots x_N \mid \mathcal{N}) = \frac{1}{N!} \prod_{x \in \mathcal{X}} \mathcal{N}(x)! \tag{4.24}$$

This is the probability mass given to the sequence's *symbol permutation*, given that its symbol histogram $\mathcal{N}$ is known. It remains to be shown how to encode $\mathcal{N}$.

Optimal encoding of the symbol histogram

If the input sequence $x_1 \ldots x_N$ is unknown, and its N symbols are independently distributed according to P, then the symbol histogram $\mathcal{N}$ is a multinomially distributed random variable:

$$\Pr(\mathcal{N} \mid P, N) = N! \prod_{x \in \mathcal{X}} \frac{P(x)^{\mathcal{N}(x)}}{\mathcal{N}(x)!} \tag{4.25}$$

Unfortunately, P itself is unknown. As before, we can place a Dirichlet prior over P:

$$\Pr(P \mid \boldsymbol{\alpha}) \;=\; \frac{\Gamma(A)}{\prod_{x \in \mathcal{X}} \Gamma(\alpha_x)} \cdot \prod_{x \in \mathcal{X}} P(x)^{(\alpha_x - 1)} \;=\; \mathrm{Dir}(P; \boldsymbol{\alpha}) \tag{4.26}$$

where $\boldsymbol{\alpha}$ is a $|\mathcal{X}|$-length parameter vector, and A is the sum of all α_x. The Dirichlet distribution is a conjugate prior to discrete finite distributions, and can be used to *learn* P from a finite number of symbol observations or counts. Furthermore, it is possible to integrate out P:

$$\int \Pr(\mathcal{N} \mid P, N) \cdot \Pr(P \mid \boldsymbol{\alpha}) \; dP \;=\; \Pr(\mathcal{N} \mid \boldsymbol{\alpha}, N) \tag{4.27}$$

yielding a compound *Dirichlet-multinomial* distribution, ranging over $|\mathcal{X}|$-length vectors of non-negative integers $\mathcal{N}(x)$ that sum to N:

$$\Pr(\mathcal{N} \mid \boldsymbol{\alpha}, N) = \frac{N!\; \Gamma(A)}{\Gamma(N + A)} \prod_{x \in \mathcal{X}} \frac{\Gamma(\mathcal{N}(x) + \alpha_x)}{\mathcal{N}(x)!\; \Gamma(\alpha_x)} \tag{4.28}$$

This distribution gives a direct connection between the histogram $\mathcal{N}$ and the hyper-parameters $\boldsymbol{\alpha}$ of the Dirichlet prior. The task of representing the distribution over symbol histograms for a given sequence length is thereby solved.

A Dirichlet-multinomial coding technique was described in section 3.4.7.

4.4.3 Which method is better?

The previous two sections derived two alternative approaches for compressing a finite sequence (of known length) from a stationary source: *online compression* and *header-payload compression*. Both approaches involve learning the unknown symbol distribution P; in the online method, this happens gradually (symbol by symbol) by addition to counts, and in the header-payload method it happens all at once and is then gradually "unlearned" by subtraction from counts. This section proves that these two approaches are in fact both optimal and have identical compression effectiveness.

Under the *header-payload method*, the joint distribution of the sequence is obtained by com-

bining equations (4.24) and (4.28):

$$\Pr(x_1 \ldots x_N, \mathcal{N} \mid \boldsymbol{\alpha}) = \left(\frac{N!\,\Gamma(A)}{\Gamma(N+A)} \prod_{x \in \mathcal{X}} \frac{\Gamma(\mathcal{N}(x)+\alpha_x)}{\mathcal{N}(x)!\,\Gamma(\alpha_x)} \right) \cdot \frac{1}{N!} \prod_{x \in \mathcal{X}} \mathcal{N}(x)! \tag{4.29}$$

The $N!$ factor cancels, as do the $\mathcal{N}(x)!$ factors, leaving the joint distribution under the header-payload method at:

$$\Pr(x_1 \ldots x_N, \mathcal{N} \mid \boldsymbol{\alpha}) = \frac{\Gamma(A)}{\Gamma(N+A)} \prod_{x \in \mathcal{X}} \frac{\Gamma(\alpha_x + \mathcal{N}(x))}{\Gamma(\alpha_x)} \tag{4.30}$$

Under the *online compression* method, the joint distribution of the entire sequence was given in equation (4.18), quoted again here:

$$\Pr(x_1 \ldots x_N \mid \boldsymbol{\alpha}, N) = \prod_{n=1}^{N} \frac{\alpha_{x_n} + \#[x_k = x]_{k=1}^{n-1}}{A + n - 1} \tag{4.18}$$

This formula can be put into a more convenient form by pulling the denominator out of the product, reparametrizing the product to range over the symbols of the alphabet (rather than indices of the sequence), and rewriting both in terms of Gamma functions:

$$\Pr(x_1 \ldots x_N \mid \boldsymbol{\alpha}, N) = \frac{\Gamma(A)}{\Gamma(N+A)} \prod_{n=1}^{N} \alpha_{x_n} + \#[x_k = x]_{k=1}^{n-1} \tag{4.31}$$

$$= \frac{\Gamma(A)}{\Gamma(N+A)} \prod_{x \in \mathcal{X}} \frac{\Gamma(\alpha_x + \mathcal{N}(x))}{\Gamma(\alpha_x)} \tag{4.32}$$

Equations (4.30) and (4.32) are equal, which means that both approaches, when implemented with arithmetic coding, compress the input sequence $x_1 \ldots x_N$ to the same number of bits.

4.5 Summary

This chapter introduced basic compression techniques for compressing sequences of independently and identically distributed symbols[6] whose symbol distribution isn't known in advance. These adaptive methods learn the unknown distribution from the sequence that is being communicated; this learning process is carried out identically by the compressor and the decompressor.

Adaptive methods are building blocks for context-sensitive sequence compressors (such as those from the PPM family), which are covered in detail in chapter 6.

Finally, the learning process of some adaptive models can be carried out in two ways: either adaptively, learning the distribution one symbol at a time, or in header-payload fashion, send-

[6] The information theory literature sometimes refers to such sequences as "sequences from a stationary memoryless source".

ing sufficient statistics of the symbol occurrence counts first, followed by the ordering of the sequence (conditional on the counts). Both approaches are mathematically equivalent. The header-payload approach separates the sequence into a symbol multiset and a permutation, each of which can be compressed independently. Multisets and permutations are combinatorial structures that differ from sequences. The following chapter derives probabilistic models and compression algorithms for a variety of fundamental combinatorial objects, and motivates a theory of structural compression.

Chapter 5

Compressing structured objects

This chapter presents techniques for compressing structured combinatorial objects, such as sets and multisets. Understanding the structural properties of the data we seek to compress is crucial for good compression. There are several basic structural objects other than sequences, some of them fundamental mathematical constructs, which have been given little attention in the compression literature. This chapter is devoted to them.

5.1 Introduction

Most data processed by electronic systems today are of a sequential nature (or are forced into sequential form), and the most common methods of data compression map sequences to shorter sequences in a lossless fashion. The processing of sequences is particularly straightforward with contemporary computers due to the sequential processing of instructions.

Due to the ubiquity of sequential data, it may appear that most objects have a *natural* sequential representation, like books or DNA sequences, but this is not generally true. Counter examples include 2-dimensional pictures, unordered entries in a database, the set of valid words in a language (for a spell checker), the histogram of letter occurrences in a document, the tree-like organisation of a file system, or the arrangement of atoms in a molecule.

Flattening structured objects into sequential representations is a common activity of computer scientists. The resulting sequences can be viewed as *encodings* that preserve the (desirable) information of the objects, allowing them to be reconstructed later. In data compression, we care not only about preserving all of the object's information in the sequence, but also about making the sequence as short as possible. This second requirement can be difficult to achieve for objects for which a sequential representation is perhaps not a natural choice.

As an example, consider the task of storing a set of N elements, where N is known to the receiver. A simple sequential representation can be created by writing out the N elements of

the set in some order. While a decoder can reconstruct the set perfectly from this sequential description, this encoding is wasteful, because each set can be described by $N!$ different sequences all representing the same set. The problem is, of course, that any such sequence stores a particular permutation of this set's elements, rather than storing *only* its elements (without the ordering). Bandwidth is wasted communicating information that isn't actually needed.

How much bandwidth is wasted exactly? There are $N!$ ways of ordering N unique items, which means that we could potentially save $\log_2(N!)$ bits $\approx (N \cdot \log_2(N) - N/\log_e 2)$ bits if the items can be encoded without communicating an order. As a concrete example, suppose we have a collection of 100 000 words of 8 letters each, and each letter has 8 bits. Storing the words naïvely in a sequence (with no separators, as they all have a fixed length) costs 800 kB. The amount of information spent on the order in which the words were sent equals $\log_2(100\,000!)$ bits $\approx 1\,516\,704$ bits $\approx 189\,588$ bytes ≈ 190 kB. This means that roughly 23% of the information in the file is wasted on the ordering.

Existing approaches

The difficulty of finding dense sequential representations for combinatorial objects has been acknowledged in the literature and various methods have been suggested.

One line of thought is *bits-back coding*, which puts wasted bandwidth to good use by filling it up with *additional data*. In the example of a set of N elements, one could send along additional information by picking one of the $N!$ possible permutations in a way that depends on other data. The idea of bits-back coding is described by Frey and Hinton (1997), Hinton and Zemel (1994), and Hinton and van Camp (1993). The bits-back approach is elegant, and can produce compactly coded representations when it has other useful data available to "plug the gaps" with. However, it doesn't really provide a solution to the problem of compactly encoding *only* the desired object.

Another relevant approach is *enumerative coding* (Cover, 1973). An enumerative coder compresses sequences from a constrained set $\mathcal{S}$ by mapping any element $s \in \mathcal{S}$ to a natural number n_s from 0 to $|\mathcal{S}|-1$ in an efficient manner. The object's number n_s can be stored compactly using an appropriate code for integers. An enumerative coder has to be derived for any given set $\mathcal{S}$. Enumerative coders produce dense uniform encodings for elements of the set they were designed for. Examples given by Cover include the subset of bit strings that have a fixed number of 1-bits, and the set of permutations of N integers. Further information on enumerative coding is given by e.g. Cleary and Witten (1984b) and Öktem (1999).

Much more generally, Varshney and Goyal (2006a,b, 2007) motivate a source coding theory for sets and multisets. They discuss factorising sequences into multiset and permutation,

and give examples where this is useful. Their generative model for multisets draws elements *iid* from a discrete distribution, recognising the rôle of the multinomial distribution and of order statistics. Furthermore, they derive theoretical results and limits regarding achievable compression rates of multiset codes. However, they do not give concrete algorithms.

Outline

This chapter introduces structural compression techniques for permutations, combinations, compositions and other combinatorial objects. These techniques form part of a structural compression toolbox from which novel compression algorithms (and sampling algorithms) can be built. Relations between the structures and corresponding compression techniques are highlighted.

Any compression algorithm implicitly defines a probabilistic model over the objects it compresses. The compression is only effective if the assumptions made by this underlying model are appropriate. For all compression techniques in this chapter, these assumptions are stated explicitly in the form of a probabilistic model. The resulting compression algorithms derived from the model are optimal if the assumptions are met.

For each type of combinatorial object, probabilistic models were carefully selected to match plausible and conceptually simple generative processes.

All of the techniques presented in this chapter, except where stated, are original work.

Notation primer

This chapter makes extensive use of multisets. A multiset $\mathcal{M}$ is an orderless collection of elements in which elements may occur multiple times. The notation $x \in \mathcal{M}$ means that an element x is contained in $\mathcal{M}$ at least once; and the multiplicity function $\mathcal{M}(x)$ denotes the exact number of times x occurs.

The cardinality of $\mathcal{M}$ is written $|\mathcal{M}|$, and counts the total number of elements including repetitions:

$$|\mathcal{M}| \;=\; \sum_{x \in \mathcal{M}} \mathcal{M}(x) \tag{5.1}$$

The *additive union* of two multisets $\mathcal{R}$ and $\mathcal{M}$ is written $\mathcal{R} \uplus \mathcal{M}$, and describes the multiset composed of all elements of $\mathcal{R}$ and all elements of $\mathcal{M}$, such that $(\mathcal{R} \uplus \mathcal{M})(x) = \mathcal{M}(x) + \mathcal{R}(x)$ for all x.

5.2 Permutations

Define a permutation to be an ordered arrangement of a *known multiset* of symbols. For example, there are 6 possible permutations of the set $\{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$:

$$(\mathtt{A}, \mathtt{B}, \mathtt{C}) \qquad (\mathtt{A}, \mathtt{C}, \mathtt{B}) \qquad (\mathtt{B}, \mathtt{A}, \mathtt{C}) \qquad (\mathtt{B}, \mathtt{C}, \mathtt{A}) \qquad (\mathtt{C}, \mathtt{A}, \mathtt{B}) \qquad (\mathtt{C}, \mathtt{B}, \mathtt{A})$$

and 3 possible permutations of the multiset $\{\mathtt{X}, \mathtt{X}, \mathtt{Y}\}$:

$$(\mathtt{X}, \mathtt{X}, \mathtt{Y}) \qquad (\mathtt{X}, \mathtt{Y}, \mathtt{X}) \qquad (\mathtt{Y}, \mathtt{X}, \mathtt{X})$$

Note that in contrast to what the above notation might suggest, a permutation itself contains no information about symbols or their occurrence counts: it just specifies an arrangement of symbols for a *given* multiset. This property distinguishes permutations from sequences: a sequence is a multiset *plus* a permutation of that multiset.

This section discusses how to encode a permutation optimally.

5.2.1 Complete permutations

Suppose we have a permutation $(x_1 \ldots x_N)$ of a multiset $\mathcal{M}$. The length N of the permutation is equal to the number $|\mathcal{M}|$ of elements in the multiset. Let $\mathcal{M}(x)$ denote the number of times a given element x occurs in $\mathcal{M}$, where $|\mathcal{M}| = \sum_{x \in \mathcal{X}} \mathcal{M}(x) = N$.

There are exactly

$$\frac{|\mathcal{M}|!}{\prod_{x \in \mathcal{X}} \mathcal{M}(x)!} \tag{5.2}$$

such permutations of $\mathcal{M}$. A uniform distribution over permutations of $\mathcal{M}$ therefore assigns to each permutation $x_1 \ldots x_N$ the following probability mass:

$$\Pr(x_1 \ldots x_N \mid \mathcal{M}) = \frac{1}{|\mathcal{M}|!} \prod_{x \in \mathcal{M}} \mathcal{M}(x)! \tag{5.3}$$

As was shown in section 4.4.2, this distribution can be factorised as follows:

$$\Pr(x_1 \ldots x_N \mid \mathcal{M}) = \prod_{n=1}^{N} \Pr(x_n \mid \mathcal{M}, \; x_1 \ldots x_{n-1}) \tag{5.4}$$

$$= \prod_{n=1}^{N} \frac{\mathcal{M}(x_n) - \sum_{k=1}^{n-1} \mathbb{1}[x_k = x_n]}{N - n + 1}, \tag{5.5}$$

yielding a component-wise encoding that is easy to interface to an arithmetic coder. Code listing 5.1 shows an encoding and decoding procedure based on this method.

Coding algorithm for permutations

ENCODING

Input: $\mathcal{M}$, N, $x_1 \ldots x_N$

1. For each symbol x_n (for $n \leftarrow 1$ to N):
   A. Compute $S(x_n) \leftarrow \sum_{x<x_n} \mathcal{M}(x)$.
   B. `storeRegion`$(S,\ S+\mathcal{M}(x_n),\ |\mathcal{M}|)$.
   C. Update $\mathcal{M} \leftarrow \mathcal{M} \setminus \{x_n{:}1\}$.

DECODING

Input: $\mathcal{M}$, N **Output:** $x_1 \ldots x_N$

1. For each symbol x_n (for $n \leftarrow 1$ to N):
   A. Compute $t \leftarrow$`getTarget`$(|\mathcal{M}|)$.
   B. Find $x_n \in \mathcal{M}$ such that $S(x_n) \leq t < S(x_n)+\mathcal{M}(x_n)$.
   C. `loadRegion`$(S,\ S+\mathcal{M}(x_n),\ |\mathcal{M}|)$
   D. Output x_n.
   E. Update $\mathcal{M} \leftarrow \mathcal{M} \setminus \{x_n{:}1\}$.

Code listing 5.1: Encoding and decoding algorithms for permutations (or truncated permutations), given advance knowledge of the symbol histogram. The multiset $\mathcal{M}$ summarises the symbol occurrences of the permutation $x_1 \ldots x_N$. The function $S(x)$ computes the cumulative symbol count of $\mathcal{M}$ (for some fixed ordering of the input alphabet $\mathcal{X}$). The update step $\mathcal{M} \leftarrow \mathcal{M} \setminus \{x_n{:}1\}$ reduces the count of element x_n by one. The external functions `storeRegion`, `loadRegion` and `getTarget` interface to an arithmetic coder, as specified in section 3.3.

Alternative methods

Methods for enumerating permutations are well known, e.g. through the work of Lehmer (1958), who provides two algorithms: `rank(`$x_1 \ldots x_N$`)`, which maps a permutation to its lexicographic index k, and the inverse algorithm `unrank(k)`, which maps the index back to a permutation.

Once a permutation's lexicographic index k is known, the index can be encoded uniformly, e.g. with an arithmetic coder:

```
storeRegion(k, k+1, factorial(N));
```

This method gives a non-sequential uniform encoding for permutations, but it is computationally impractical due to its explicit computation of the factorial.[1]

A good survey on permutations is given in (Knuth, 1998), and further details are given in (Knuth, 2004).

[1] For example, a 64-bit integer cannot represent $N!$ for any $N > 20$.

5.2.2 Truncated permutations

The permutations discussed so far were all of length $|\mathcal{M}|$, obtained by drawing a sequence of elements from $\mathcal{M}$ without replacement until no elements are left. More generally, suppose that we draw a sequence of elements from $\mathcal{M}$ without replacement, but we stop after K draws. The result is a truncated permutation $x_1 \ldots x_K$ of length K, called a *K-permutation*. (And K should be $\leq |\mathcal{M}|$.)

The probability of drawing a given K-permutation from $\mathcal{M}$ is:

$$\Pr(x_1 \ldots x_K \mid K, \mathcal{M}) \;=\; \frac{(|\mathcal{M}| - |\mathcal{K}|)!}{|\mathcal{M}|!} \prod_{x \in \mathcal{M}} \frac{\mathcal{M}(x)!}{(\mathcal{M}(x) - \mathcal{K}(x))!} \tag{5.6}$$

where $\mathcal{K}$ is the submultiset of $\mathcal{M}$ summarising how often any given value x occurs in the sequence $x_1 \ldots x_K$.

To encode or decode a truncated permutation, the same technique can be used as for complete permutations, but stopping after K symbols. The algorithm encodes each K-permutation optimally, i.e. with length proportional to the logarithm of its occurrence probability. For truncated permutations, the occurrence probability isn't uniform, except when $\mathcal{M}$ contains no repetitions, or when $K = |\mathcal{M}|$.

5.2.3 Set permutations via elimination sampling

A way of generating permutations of unique elements can be defined through an incremental process of *sampling with elimination* from a discrete distribution D. Each time an element is drawn from an elimination sampling process on D, it is eliminated and can never be sampled again. Such a process produces sequences of unique elements from the support of D; the uniqueness property makes these sequences permutations of a *set* (rather than a multiset).

The difference from other sampling schemes can be explained as follows. Consider an urn that contains balls of various colours. Sampling with replacement draws a ball, notes its colour and places the ball back in the urn. Sampling without replacement draws a ball, notes its colour and throws the ball away. Sampling with elimination draws a ball, notes its colour and removes all balls of that colour from the urn. For any urn scheme, sampling must finish when no balls are left in the urn. When sampling with elimination is applied to a discrete distribution D over some set $\mathcal{X}$, the urn will be empty after each unique value has been drawn.

Suppose a sequence of values $x_1 \ldots x_K$ is obtained by making K consecutive draws from D with elimination, where $K \leq |\mathcal{X}|$. The result is a permutation of K unique values of $\mathcal{X}$, whose ordering is influenced by the probability mass of each element.

Formally, the distribution over such permutations $x_1 \ldots x_K$ is:

$$\Pr(x_1 \ldots x_K \mid D) = \prod_{k=1}^{K} D_{\setminus\{x_1 \ldots x_{k-1}\}}(x_k) \tag{5.7}$$

where $D_{\setminus\mathcal{R}}$ denotes the distribution that derives from D by excluding a subset $\mathcal{R}$ of its domain; this notation is defined in section 3.5.2. A permutation obtained through an elimination sampling process can be optimally compressed using exclusion coding. The first element x_1 is encoded with probability mass $D(x_1)$, the second element with probability mass $D_{\setminus\{x_1\}}(x_2)$ and the third with $D_{\setminus\{x_1,x_2\}}(x_3)$, etc.

Note that sequential draws from this elimination sampling process are not generally exchangeable: different permutations of the same elements may have different probability mass. The permutation with least probability mass is produced by drawing the least probable element at every step. In the special case that D is a uniform distribution, every permutation has equal probability mass of $\frac{1}{|\mathcal{X}|!}$.

5.3 Combinations

Permutations can be viewed as fixed-length sequences generated by drawing elements from a given multiset without replacement. If the order of a permutation is thrown away, the result is a *combination*. A combination is a way of choosing a fixed-size multiset from a known larger multiset.

Given a multiset $\mathcal{M}$, any K-sized submultiset $\mathcal{K} \subseteq \mathcal{M}$ is called a K-combination of $\mathcal{M}$. One can think of a K-combination as a K-permutation without the ordering. (K-permutations were defined in section 5.2.2.)

For example, there are 4 possible 2-combinations of the multiset $\{\mathtt{A}, \mathtt{B}, \mathtt{B}, \mathtt{C}\}$:

$$\{\mathtt{A}, \mathtt{B}\} \quad \{\mathtt{A}, \mathtt{C}\} \quad \{\mathtt{B}, \mathtt{B}\} \quad \{\mathtt{B}, \mathtt{C}\}$$

Compare this to the 7 possible 2-permutations of the same multiset:

$$(\mathtt{A}, \mathtt{B}) \quad (\mathtt{A}, \mathtt{C}) \quad (\mathtt{B}, \mathtt{A}) \quad (\mathtt{B}, \mathtt{B}) \quad (\mathtt{B}, \mathtt{C}) \quad (\mathtt{C}, \mathtt{A}) \quad (\mathtt{C}, \mathtt{B})$$

When K-combinations are created by randomly drawing K elements from a multiset $\mathcal{M}$ with repetitions, the resulting distribution is not generally uniform. For instance, in the example given above, the chance of drawing combination $\{\mathtt{B}, \mathtt{C}\}$ is higher than drawing $\{\mathtt{A}, \mathtt{C}\}$. An extensive treatment of algorithms for generating combinations is given by Knuth (2005a).

One way of computing the probability mass of a given K-combination $\mathcal{K}$ drawn from $\mathcal{M}$ is to sum up the probability masses of all K-permutations of $\mathcal{M}$ whose occurrence counts match $\mathcal{K}$.

We know the number of permutations for any given multiset $\mathcal{K}$ equals:

$$\frac{|\mathcal{K}|!}{\prod_{x \in \mathcal{K}} \mathcal{K}(x)!} \tag{5.8}$$

We also know from equation (5.6) that the probability mass of a K-permutation drawn from $\mathcal{M}$ is order-invariant and depends only on its occurrence counts $\mathcal{K}(x)$. Summing up the probabilities of all K-permutations therefore reduces to the product of the right hand side of equation (5.6) and the quantity (5.8).

The probability of a given K-combination $\mathcal{K}$, drawn from a multiset $\mathcal{M}$, is therefore:

$$\Pr(\mathcal{K} \mid \mathcal{M}) = \frac{(|\mathcal{M}| - |\mathcal{K}|)!\ |\mathcal{K}|!}{|\mathcal{M}|!} \cdot \prod_{x \in \mathcal{K}} \frac{\mathcal{M}(x)!}{(\mathcal{M}(x) - \mathcal{K}(x))!\ \mathcal{K}(x)!} \tag{5.9}$$

$$= \binom{|\mathcal{M}|}{|\mathcal{K}|}^{-1} \cdot \prod_{x \in \mathcal{K}} \binom{\mathcal{M}(x)}{\mathcal{K}(x)} \tag{5.10}$$

In the case when $\mathcal{M}$ is a strict set, i.e. $\mathcal{M}(x) = 1$ for all $x \in \mathcal{M}$, the expression reduces to a uniform distribution, as all valid combinations $\mathcal{K}$ are then equally likely.

To derive a compression method for K-combinations, the distribution in (5.9) and (5.10) can be factorised into univariate distributions that interface more easily to an arithmetic coder. These univariate distributions should range over quantities that can be easily computed from $\mathcal{K}$, such as the component multiplicities $\mathcal{K}(x)$. One suitable approach can be derived from the quotient of two nested combinations:

$$\frac{\Pr(\mathcal{K} \mid \mathcal{M})}{\Pr\left(\mathcal{K}_{\setminus\{x\}} \,\middle|\, \mathcal{M}_{\setminus\{x\}}\right)} = \binom{|\mathcal{M}|}{|\mathcal{K}|}^{-1} \binom{|\mathcal{M}| - \mathcal{M}(x)}{|\mathcal{K}| - \mathcal{K}(x)} \binom{\mathcal{M}(x)}{\mathcal{K}(x)} = \Pr(\mathcal{K}(x) \mid \mathcal{M}, |\mathcal{K}|) \tag{5.11}$$

where $\mathcal{M}_{\setminus\{x\}}$ is shorthand for the multiset $\mathcal{M}$ with all occurrences of x removed. This insight suggests the following recursive factorisation:

$$\Pr\left(\mathcal{K} \,\middle|\, \mathcal{M}\right) = \Pr\left(\mathcal{K}(x) \,\middle|\, \mathcal{M}, |\mathcal{K}|\right) \cdot \Pr\left(\mathcal{K}_{\setminus\{x\}} \,\middle|\, \mathcal{M}_{\setminus\{x\}}\right) \tag{5.12}$$

Any K-combination that's distributed according to (5.9) is therefore optimally encoded with a component-wise sequence of univariate arithmetic coding steps, using equation (5.12). I will call this coding technique a *multiset combination code*.

5.4 Compositions

An *integer composition* of a natural number N is a sequence of non-negative integers $(n_1 \ldots n_K)$ that sum to N. It is common to impose further constraints on such compositions, for example by requiring each component n_k to be non-zero (in strictly positive compositions), or by

limiting the permitted number of components to a fixed number K (in K-compositions).

Strictly positive compositions find use in the encoding of ordered multiset partitions (described in section 5.6), and K-compositions can be used to concisely represent the structure of a multiset of known size and domain (described in section 5.5).

5.4.1 Strictly positive compositions with unbounded components

Call a composition *strictly positive* when all of its components are non-zero. For any positive integer N, there are exactly 2^{N-1} strictly positive integer compositions.[2] For example, there are eight strictly positive compositions of the number 4:

$$\begin{array}{cccc} (4) & (3,1) & (2,2) & (2,1,1) \\ (1,3) & (1,2,1) & (1,1,2) & (1,1,1,1) \end{array}$$

For a uniform distribution over strictly positive compositions (of a given N), the following method provides an optimal encoding scheme: for each component n_k in order, write $(n_k - 1)$ `1`-bits, followed by a `0`-bit (except for the last component n_K, whose `0`-bit is omitted). This method encodes any composition using exactly $N - 1$ bits.

5.4.2 Compositions with fixed number of components

A composition $(n_1 \ldots n_K)$ with K components is called a K-composition of N. The components of a K-composition may be zero or positive integers, and must sum to N. For example, there are fifteen 3-compositions of the number 4:

$$\begin{array}{ccccc} (4,0,0) & (3,1,0) & (0,3,1) & (2,2,0) & (2,1,1) \\ (0,4,0) & (3,0,1) & (1,0,3) & (2,0,2) & (1,2,1) \\ (0,0,4) & (1,3,0) & (0,1,3) & (0,2,2) & (1,1,2) \ . \end{array}$$

For a given N, the total number of possible K-compositions equals:

$$C_K(N) = \binom{N+K-1}{K-1} = \frac{(N+K-1)!}{N!\ (K-1)!} \tag{5.13}$$

If N and K are known, how does one optimally compress a K-composition of N? The next two subsections present two alternative answers.

[2]Proof: the unary representation of N is a sequence of N `1`-digits. There are $N - 1$ gaps between the unary digits, and 2^{N-1} ways of writing either a plus (+) or a comma (,) into each gap. Each configuration corresponds to one unique strictly positive composition of N.

5.4.3 Uniform K-compositions

Assuming that we want the choice to be uniform, we should assign each K-composition a probability mass of:

$$\Pr(n_1 \ldots n_K \mid N, K) = \frac{N!\ (K-1)!}{(N+K-1)!} \tag{5.14}$$

One way of building such a coding scheme is to enumerate all $C_K(N)$ of the K-compositions, find the index c of the chosen composition in that list, and encode c as a uniform integer between 0 and $C_K(N)-1$. The decoder can recover the chosen K-composition by generating the same list, decoding integer index c and looking up position c in the list.

An alternative is to encode the K-composition $\boldsymbol{n}$ component-wise, in such a way that the resulting probability is still uniform. This is not as easy as it might first appear: for example, encoding each component n_k with a discrete uniform distribution over the allowed range of remaining values will give a non-uniform and order-dependent result.

Component-wise encoding can be done correctly by following the recursive structure of the composition. The number of possible K-compositions depends on the value of n_1. For example, if $n_1 = N$, then the remaining components $n_2 \ldots n_K$ must be zero. More generally, when n_1 is fixed, the remaining components can form exactly $C_{K-1}(N-n_1)$ valid $(K-1)$-compositions. We can assign probability mass to each possible value of n_1 in proportion to the number of compositions that can be formed from the remaining components, resulting in the following probability for n_1:

$$\Pr(n_1 \mid N, K) = \frac{C_{K-1}(N-n_1)}{C_K(N)} \tag{5.15}$$

Applying this rule recursively yields a general expression for the probability mass of the kth component, conditional upon the preceding components:

$$\Pr(n_k \mid n_1 \ldots n_{k-1}, N, K) = \frac{C_{K-k}(N-\sum_{j=1}^{k} n_j)}{C_{K-k+1}(N-\sum_{j=1}^{k-1} n_j)} \tag{5.16}$$

This form lends itself to a simple sequential encoding procedure that can be interfaced to an arithmetic coder. Observe that the joint probability of $n_1 \ldots n_K$ equals the probability mass given in (5.14), so the result is indeed uniform.

5.4.4 Multinomial K-compositions

A uniform distribution over K-compositions may not be a natural choice. For example, consider a generative process in which N outcomes $x_1 \ldots x_N$ are drawn from a uniform distribution over the values $\{1 \ldots K\}$. The N outcomes can be summarised by a histogram $(n_1 \ldots n_K)$ that indicates how often each of the K possible outcomes occurred. The components n_k sum to N, so the histogram $(n_1 \ldots n_K)$ forms a K-composition.

K-compositions obtained this way are clearly not uniformly distributed. Instead, their probability mass is given by a multinomial distribution. For N elements drawn from a discrete uniform distribution over K elements, the K-composition has probability:

$$\Pr(n_1 \ldots n_K \mid N, K) = \frac{N!}{K^N} \cdot \prod_{k=1}^{K} \frac{1}{n_k!} \tag{5.17}$$

This distribution can also be factorised into a component-wise encoding, giving:

$$\Pr(n_1 \ldots n_K \mid N, K) = \prod_{k=1}^{K} \Pr(n_k \mid n_1 \ldots n_{k-1}, N, K) \tag{5.18}$$

$$= \prod_{k=1}^{K} \binom{N - \sum_{j=1}^{k-1} n_j}{n_k} \frac{1}{K^{n_k}} \tag{5.19}$$

These expressions can be generalised to the case where the N draws are made from any discrete distribution P over integers $\{1 \ldots K\}$. A K-composition created this way has the following distribution:

$$\Pr(n_1 \ldots n_K \mid N, K, P) = N! \cdot \prod_{k=1}^{K} \frac{P(k)^{n_k}}{n_k!} \tag{5.20}$$

And it factorises into a product of binomial distributions:

$$\Pr(n_1 \ldots n_K \mid N, K, P) = \prod_{k=1}^{K} \binom{N - \sum_{j=1}^{k-1} n_j}{n_k} \cdot P(k)^{n_k} \tag{5.21}$$

This result defines an optimal sequential encoding / decoding procedure for multinomial compositions, using the binomial code from section 3.4.4. This *multinomial composition code* forms the basis of several of the multiset coding techniques in section 5.5.

5.5 Multisets

Multisets can be viewed as a natural generalisation of sets, in which elements are allowed to occur more than once. For example, the multiset $\{\mathtt{A}, \mathtt{B}, \mathtt{B}\}$ is the collection which contains one occurrence of symbol `A` and two occurrences of symbol `B`. The same multiset may also be written $\{\mathtt{A}{:}1, \mathtt{B}{:}2\}$.

Just as set membership can be viewed as a function of type $\mathcal{X} \to \{0, 1\}$, multisets have an associated *multiplicity function* of type $\mathcal{X} \to \mathbb{N}$ indicating the number of times any given element occurs. The notation $\mathcal{M}(x)$ denotes the number of times x occurs in multiset $\mathcal{M}$. For compatibility with set notation, $x \in \mathcal{M}$ means $\mathcal{M}(x) > 0$, and $x \notin \mathcal{M}$ means $\mathcal{M}(x) = 0$.

The cardinality of a multiset is defined as:

$$|\mathcal{M}| = \sum_{x \in \mathcal{X}} \mathcal{M}(x). \tag{5.22}$$

Multisets occur naturally in data compression: for example, the histogram of letter occurrences in a document is described by a multiset. Many advanced compression methods accumulate symbol counts for each of a large collection of contexts – each context effectively stores a multiset. It is therefore interesting to investigate how multisets themselves can be compressed.

In contrast to a sequence, a multiset contains no information about the order of the symbols it contains. It was shown in section 4.4 that a sequence can be split into a multiset and a permutation, and how these two components can be compressed separately in an effective way. This section elaborates on the compression of multisets, given various different generative assumptions.

The approaches presented here arithmetically encode a multiset $\mathcal{M}$ by first sending its cardinality $|\mathcal{M}|$, and then its occurrence counts $\mathcal{M}(x)$ for all $x \in \mathcal{X}$, conditional on $|\mathcal{M}|$.

5.5.1 Multisets via repeated draws from a known distribution

One way of creating a multiset is by drawing its elements *iid* from some discrete distribution D. Suppose that D ranges over a finite set of elements $\mathcal{X}$. We could form a multiset of size N by taking N independent samples $x_1 \ldots x_N$ from D. Let the size N be distributed according to some chosen distribution L over positive integers. The generative process for such a multiset $\mathcal{M}$ can be written as follows:

$$N \sim L \tag{5.23}$$

$$x_1 \ldots x_N \overset{\text{iid}}{\sim} D \tag{5.24}$$

$$\text{and creating } \mathcal{M} := \biguplus_{n=1}^{N} \{x_n\} \tag{5.25}$$

where $\uplus$ denotes multiset-union. Because D is discrete, elements x_n drawn from it can repeat with non-zero probability. $\mathcal{M}(x)$ counts how often any given value x occurred in the N draws $x_1 \ldots x_N$ from D. The sum of occurrences equals $N = |\mathcal{M}|$.

This generative model assigns to any multiset $\mathcal{M}$ the probability mass of:

$$\Pr(\mathcal{M} \mid L, D) = L(|\mathcal{M}|) \cdot |\mathcal{M}|! \cdot \prod_{x \in \mathcal{M}} \frac{D(x)^{\mathcal{M}(x)}}{\mathcal{M}(x)!} \tag{5.26}$$

The occurrence counts $\mathcal{M}(x)$ form a $|\mathcal{X}|$-composition of $|\mathcal{M}|$, for any total ordering on $\mathcal{X}$. This suggests the following optimal encoding (and decoding) procedure for a multiset of this

kind:

1. Encode (or decode) $|\mathcal{M}|$ using a code for the size distribution L.
2. For some fixed ordering on $\mathcal{M}$, encode (or decode) the counts $\mathcal{M}(x)$ using the multinomial composition code (from section 5.4.4) with parameters $K = |\mathcal{X}|$, $N = |\mathcal{M}|$ and prior D.

This method is optimal assuming that the coding method for L is optimal. Its operation requires $\mathcal{X}$ to be a finite and enumerable set.

5.5.2 Multisets drawn from a Poisson process

A Poisson process can be viewed as a special case of the model presented in the previous section. Poisson processes model the size of the multiset with a Poisson distribution:

$$|\mathcal{M}| \sim \text{Poisson}(\lambda) \tag{5.27}$$

where $\lambda \in \mathbb{R}^+$ is a non-zero rate parameter. The Poisson processes used here are parametrised by a rate λ and the discrete distribution D over $\mathcal{X}$, and assign probability mass to multisets $\mathcal{M}$ as follows:

$$\Pr(\mathcal{M} \mid D, \lambda) = \frac{e^{-\lambda}\lambda^{|\mathcal{M}|}}{|\mathcal{M}|!} \cdot |\mathcal{M}|! \cdot \prod_{x \in \mathcal{M}} \frac{D(x)^{\mathcal{M}(x)}}{\mathcal{M}(x)!} \tag{5.28}$$

The Poisson distribution isn't an arbitrary choice: its use gives the resulting multiset particularly nice properties, especially when it comes to encoding. General arguments for the usefulness and ubiquity of Poisson processes are given by Kingman (1993).[3]

Of course the sampling, encoding and decoding procedures from the previous section can be used as before. But exploiting a property of Poisson processes, one can derive a simpler encoding / decoding method.

Observe that the Poisson distribution distributes over the product in the multinomial distribution:

$$\underbrace{\frac{e^{-\lambda}\lambda^{|\mathcal{M}|}}{|\mathcal{M}|!}}_{\text{Poisson}(\lambda)} \cdot \underbrace{|\mathcal{M}|! \prod_{x \in \mathcal{X}} \frac{D(x)^{\mathcal{M}(x)}}{\mathcal{M}(x)!}}_{\text{Mult}(|\mathcal{M}|, D)} = \prod_{x \in \mathcal{X}} \underbrace{\frac{e^{-\lambda D(x)}\lambda^{\mathcal{M}(x)} D(x)^{\mathcal{M}(x)}}{\mathcal{M}(x)!}}_{\text{Poisson}(\lambda \cdot D(x))} \tag{5.30}$$

[3] Poisson processes are typically constructed over continuous measures μ which have no atomic components. In that case elements do not repeat (with probability 1), and the multinomial selection factor reduces to $|\mathcal{M}|!$, giving the more familiar form:

$$\Pr(\mathcal{M} \mid \mu, \lambda) = \frac{e^{-\lambda}\,\lambda^{|\mathcal{M}|}}{|\mathcal{M}|!} \cdot \prod_{x \in \mathcal{M}} \mu(x)^{\mathcal{M}(x)} \tag{5.29}$$

In a compression setting, however, pretty much all interesting measures are atomic, as non-atomic (zero-mass) elements cannot be represented with a finite number of bits.

This is because:

$$\prod_{x \in \mathcal{X}} e^{-\lambda D(x)} = e^{-\lambda} \qquad (5.31) \qquad \text{and} \qquad \prod_{x \in \mathcal{X}} \lambda^{\mathcal{M}(x)} = \lambda^{|\mathcal{M}|}. \qquad (5.32)$$

Therefore each $\mathcal{M}(x)$ is individually Poisson distributed:

$$\mathcal{M}(x) \sim \text{Poisson}(\lambda D(x)) \qquad (5.33)$$

A Poisson-process-distributed multiset $\mathcal{M}$ can therefore be conveniently encoded and decoded with the following algorithm:

1. For each $x \in \mathcal{X}$ in some predetermined sequential order, independently encode (or decode) $\mathcal{M}(x)$ using a Poisson code with mean $\lambda D(x)$.

A nice consequence of this choice of representation is that it's easily possible to store submultisets: one only needs to encode $\mathcal{M}(x)$ for those x one cares about. It can also be used to store a finite submultiset when $\mathcal{X}$ and D are countably infinite.

5.5.3 Multisets from unknown discrete distributions

So far we've created multisets by making repeated draws from a distribution D, assuming that D is known to both the encoder and decoder. Let's now assume that a multiset $\mathcal{M}$ is created by drawing from an *unknown* distribution D with a discrete finite domain $\mathcal{X}$. As in section 4.4, we'll model the uncertainty of D with a Dirichlet distribution:

$$\begin{aligned} D &\sim \text{Dir}(\boldsymbol{\alpha}) \\ x_1 \ldots x_N &\overset{\text{iid}}{\sim} D \end{aligned} \qquad (5.34)$$

where $\boldsymbol{\alpha}$ is an $|\mathcal{X}|$-dimensional Dirichlet parameter vector. The number of elements N is again distributed according to some independent distribution L over positive integers.

Integrating out D from the product of equation (5.26) and the Dirichlet prior, the probability of multiset $\mathcal{M}$ becomes:

$$\Pr(\mathcal{M} \mid L, \boldsymbol{\alpha}) = L(|\mathcal{M}|) \cdot \int \text{Mult}(\mathcal{M} \mid D) \cdot \text{Dir}(D \mid \boldsymbol{\alpha})\, dD \qquad (5.35)$$

$$= L(|\mathcal{M}|) \cdot \frac{|\mathcal{M}|!\, \Gamma(A)}{\Gamma(|\mathcal{M}| + A)} \prod_{x \in \mathcal{X}} \frac{\Gamma(\mathcal{M}(x) + \alpha_x)}{\mathcal{M}(x)!\, \Gamma(\alpha_x)} \qquad (5.36)$$

where A is the sum of the components of $\boldsymbol{\alpha}$. The result is a product of a compound Dirichlet-multinomial distribution over $\mathcal{M}$'s occurrence counts, and the distribution of $\mathcal{M}$'s size.

The multiset $\mathcal{M}$, conditional on $\boldsymbol{\alpha}$ and L, can then be encoded by first sending its length using a code for L, and then an $|\mathcal{X}|$-length vector of counts $\mathcal{M}(x)$ using a Dirichlet-multinomial code (such as the one described in section 3.4.7).

5.5.4 Submultisets

We considered generative processes for multisets in which elements were drawn *iid* from some distribution. Drawing *iid* means sampling with replacement, which makes the multiplicities in the resulting multiset representative of the source distribution.

A multiset can also arise as a submultiset of a larger multiset $\mathcal{M}$. Consider the following generative process: to obtain a random submultiset $\mathcal{K} \subseteq \mathcal{M}$ of given size K, draw elements uniformly from $\mathcal{M}$ without replacement. [The idea is superficially similar to drawing a permutation of $\mathcal{M}$ (like in section 5.2), which is obtained by repeatedly removing from $\mathcal{M}$ at random and recording the order. This time we don't care about the order.]

The resulting draws, when taken as an unordered collection, form a K-combination of $\mathcal{M}$, and can be compressed optimally using the multiset combination code of section 5.3.

5.5.5 Multisets from a Blackwell–MacQueen urn scheme

As an alternative to forming a multiset by drawing elements *iid* from some probability distribution, let's consider drawing elements from the urn scheme of Blackwell and MacQueen (1973). This urn scheme produces exchangeable sequences of elements such that the resulting distribution over elements is a Dirichlet process (as introduced in section 4.2.2).

A Dirichlet process prior (DP prior) can be viewed as an infinite-dimensional generalisation of the Dirichlet distribution, and can be used to learn distributions with domains that are countably infinite. The Blackwell–MacQueen construction works as follows:

$$x_{N+1} \mid x_1 \ldots x_N \;\sim\; \frac{\alpha}{N+\alpha} H(x) + \sum_{n=1}^{N} \frac{1}{N+\alpha} \delta_{x_n} \tag{5.37}$$

where α is a concentration parameter and H is a base distribution. The joint distribution of the next K draws from this urn scheme, conditional upon the previous N draws, is

$$\Pr(x_{N+1} \ldots x_{N+K} \mid H, \alpha;\, x_1 \ldots x_N) \;=\; \prod_{k=1}^{K} \Pr(x_{N+k} \mid H, \alpha;\, x_1 \ldots x_{N+k-1}), \tag{5.38}$$

which simplifies to:

$$= \prod_{k=1}^{K} \left(\frac{\#[x_m = x_{N+k}]_{m=1}^{N+k-1}}{N+k-1+\alpha} + \frac{\alpha}{N+k-1+\alpha} H(x_{N+k}) \right) \tag{5.39}$$

$$= \frac{(N-1+\alpha)!}{(N+K-1+\alpha)!} \cdot \prod_{k=1}^{K} \left(\#[x_m = x_{N+k}]_{m=1}^{N+k-1} + \alpha H(x_{N+k}) \right) \tag{5.40}$$

$$= \frac{\Gamma(N+\alpha)}{\Gamma(N+K+\alpha)} \cdot \prod_{x \in \mathcal{X}} \frac{\Gamma(\alpha H(x) + \mathcal{N}(x) + \mathcal{K}(x))}{\Gamma(\alpha H(x) + \mathcal{N}(x))}. \tag{5.41}$$

Because sequences from the Blackwell–MacQueen urn are infinitely exchangeable, the joint probability (5.41) depends only on the multisets $\mathcal{N}$ and $\mathcal{K}$ that summarise the element occurrences in the subsequences $(x_1 \ldots x_N)$ and $(x_{N+1} \ldots x_{N+K})$, respectively:

$$\begin{aligned} \mathcal{N}(x) &\stackrel{\text{def}}{=} \#[x_n = x]_{n=1}^{N} \\ \mathcal{K}(x) &\stackrel{\text{def}}{=} \#[x_{N+k} = x]_{k=1}^{K} \end{aligned} \tag{5.42}$$

where $N = \sum_{x \in \mathcal{X}} \mathcal{N}(x)$ and $K = \sum_{x \in \mathcal{X}} \mathcal{K}(x)$. Just as the counts $\mathcal{N}$ are sufficient statistics of the urn's history, the $\mathcal{K}$ are sufficient statistics of the urn's future (for the next K elements). We will now derive the distribution over $\mathcal{K}$ given $\mathcal{N}$ (and DP prior parameters α and H).

Equation (5.41) gives the joint probability mass of a *sequence* of the next K draws. We can marginalise out the order of the sequence by summing over all permutations: the probability of the multiset $\{x_{N+1} \ldots x_{N+K}\}$ equals the sum of the probabilities of all sequences that can be formed from the multiset's elements.

Because the joint probability of each permutation depends only on the multisets $\mathcal{N}$ and $\mathcal{K}$, all permutations (of any fixed $\mathcal{K}$) have equal mass, and the sum turns into a simple product:

$$\Pr(\mathcal{K} \mid K, H, \alpha;\, x_1 \ldots x_N) = \frac{K!}{\prod_{x \in \mathcal{X}} \mathcal{K}(x)!} \cdot \Pr(x_{N+1} \ldots x_{N+K} \mid H, \alpha;\, x_1 \ldots x_N). \tag{5.43}$$

The factor in front of the right hand side of equation (5.43) is the multinomial coefficient, i.e. the total number of possible permutations as given by equation (5.2). Combining equations (5.41) and (5.43) yields the following distribution over $\mathcal{K}$:

$$\Pr(\mathcal{K} \mid \mathcal{N}, K, H, \alpha) = \frac{K! \cdot \Gamma(N+\alpha)}{\Gamma(N+K+\alpha)} \cdot \prod_{x \in \mathcal{X}} \frac{\Gamma\big(\alpha H(x) + \mathcal{N}(x) + \mathcal{K}(x)\big)}{\mathcal{K}(x)! \cdot \Gamma\big(\alpha H(x) + \mathcal{N}(x)\big)} \tag{5.44}$$

This result can be used to compress multisets $\mathcal{K}$ whose underlying distribution is partially known (through $\mathcal{N}$ and base distribution H).

However, to compress such a multiset with an arithmetic coder, the multivariate distribution (5.44) must be factorised into a product of univariate distributions. Fortunately, the probability of $\mathcal{K}$ can be split into probabilities of each count $\mathcal{K}(x)$ for each $x \in \mathcal{X}$ in some predetermined order, conditioned on all counts preceding the current one. This approach is similar to the Dirichlet-multinomial coding scheme of section 3.4.7. Each of the conditional distributions depends on the following known quantities: the remaining elements $\mathcal{R}_x$ from $\mathcal{X}$,

the remaining number of draws R_k not yet accounted for, and quantities derived from the counts $\mathcal{N}$. In particular, R_n is the sum of all $\mathcal{N}(x)$ for those x in $\mathcal{R}_x$. Similarly, $\mathcal{A}_x = \mathcal{R}_x \cup \{x\}$ and $A_k = R_k + \mathcal{K}(x)$.

$$\Pr\Big(\mathcal{K}(x) \,\Big|\, \mathcal{R}_x, R_k, R_n, H, \alpha, \mathcal{N}\Big) = \frac{A_k!}{R_k!\;\mathcal{K}(x)!} \cdot \frac{\Gamma\Big(\alpha H(x) + \mathcal{N}(x) + \mathcal{K}(x)\Big)}{\Gamma\Big(\alpha H(x) + \mathcal{N}(x)\Big)} \cdot \frac{\Gamma\Big(\alpha H(\mathcal{A}_x) + R_n\Big)}{\Gamma\Big(\alpha H(\mathcal{A}_x) + R_n + R_k\Big)} \cdot \frac{\Gamma\Big(\alpha H(\mathcal{R}_x) + R_n + R_k\Big)}{\Gamma\Big(\alpha H(\mathcal{R}_x) + R_n\Big)} \tag{5.45}$$

Observe that the structure of equation (5.45) corresponds closely to that of equation (5.44). A convenient component-wise sequential encoding is thereby established.

5.6 Ordered partitions

An *ordered partition* of a multiset $\mathcal{M}$ is a sequence of submultisets $\mathcal{A}_1$ ... $\mathcal{A}_K$ whose additive union equals $\mathcal{M}$:

$$\biguplus_{k=1}^{K} \mathcal{A}_k \;=\; \mathcal{M} \tag{5.46}$$

For example, there are eight possible ordered partitions of the multiset $\{\bullet, \bullet, \circ\}$:

$\{\bullet\}, \{\bullet\}, \{\circ\}$
$\{\bullet\}, \{\circ\}, \{\bullet\}$
$\{\circ\}, \{\bullet\}, \{\bullet\}$

$\{\bullet, \bullet\}, \{\circ\}$
$\{\bullet, \circ\}, \{\bullet\}$
$\{\bullet\}, \{\bullet, \circ\}$
$\{\circ\}, \{\bullet, \bullet\}$

$\{\bullet, \bullet, \circ\}$

Ordered partitions can be seen as a natural hybrid of a sequence and a multiset, where some order information of a collection of elements is preserved, and some is not. An ordered partition with only one block of size $|\mathcal{M}|$ corresponds to a normal multiset, and an ordered partition with $|\mathcal{M}|$ blocks of size 1 corresponds to a normal sequence.

Compressing an ordered partition, given $\mathcal{M}$, can be done by combining techniques developed in earlier sections. A suitable algorithm could work as follows:

(I) Encode the sequence of block sizes $|\mathcal{A}_1|$ ... $|\mathcal{A}_K|$ where the number K of blocks may be known or unknown in advance. Since the sum of the $|\mathcal{A}_k|$ equals $|\mathcal{M}|$, the sequence of block sizes form a composition of $|\mathcal{M}|$ and can be encoded using any of the composition codes from section 5.4.

(II) Sequentially encode each block $\mathcal{A}_k$ of the partition. For each $\mathcal{A}_k$, it is true that:

$$\mathcal{A}_k \subseteq \mathcal{M} \setminus \biguplus_{j<k} \mathcal{A}_j \tag{5.47}$$

Therefore each block $\mathcal{A}_k$ can be encoded as an $|\mathcal{A}_k|$-sized submultiset of the remaining elements, using the multiset combination code of section 5.3. Note that no bits are used for the last block, as it has probability 1.

The procedure can be amended depending on what information is available to the encoder and decoder in advance. For example:

- If the sequence of block sizes $|\mathcal{A}_1|$ ... $|\mathcal{A}_K|$ of the ordered partition is known in advance, both encoder and decoder can skip directly to step (II).
- If just the multiset of block sizes $\{|\mathcal{A}_1|, \cdots, |\mathcal{A}_K|\}$ is known in advance, the encoder only needs to transmit its permutation, e.g. using the permutation coder from section 5.2.
- If only K is known, a K-composition code can be used in step (I), such as the one from section 5.4.3. If K is unknown, it can either be transmitted separately using a discrete code over integers $\{1, \cdots, |\mathcal{M}|\}$, or encoded implicitly using the uniform composition code given at the beginning of section 5.4.

A general derivation of the number of multiset partitions is given by Bender (1974). Algorithms for generating sorted and unsorted set partitions can be found in (Knuth, 2005b).

5.7 Sequences

Sequences are perhaps the most widespread and familiar structural form of data on contemporary computer systems. Compression methods for sequences are well known: most compression algorithms in the literature are sequence compressors.

Since data on contemporary computer systems are stored as sequences of bytes, sequence compressors are in common use and often applied to any type of data found on a computer system. State of the art sequence models do not generally assume that elements of the sequence are independently distributed; typically, a collection of context-dependent conditional distributions is learned from the data and used to compress subsequent symbols. An analysis of state of the art sequence compression is described in chapter 6.

Remaining consistent with the spirit of this chapter, this section explores only fundamental structural properties of sequences, their relations to other combinatorial objects, and generative processes for producing them.

5.7.1 Sequences of known distribution, known length

Let's assume a finite sequence of elements $x_1 \ldots x_N$ is drawn *iid* from a known discrete distribution D over some space $\mathcal{X}$. The joint probability of all elements in the sequence is:

$$\Pr(x_1 \ldots x_N \mid N, D) = \prod_{n=1}^{N} D(x_n) \tag{5.48}$$

If we want to compress this sequence, assuming that its length N is known in advance to both the encoder and decoder, the elements can simply be encoded sequentially using a code for D. Since the elements are independently and identically distributed, the last element of the sequence is compressed in the same way as the first; the length N and distribution D are the only useful pieces of information for compressing the sequence, the rest is unpredictable randomness.

An alternative to the sequential coding method given above is to factorise the sequence into a multiset and a permutation, each of which can be encoded separately using the structural compression techniques from earlier parts of this chapter.
Define the multiset $\mathcal{M}$ to be the occurrence counts of elements in $x_1 \ldots x_N$:

$$\mathcal{M}(x) \;=\; \#[x_n = x]_{n=1}^{N} \tag{5.49}$$

The total count $|\mathcal{M}|$ equals N. Encoding the sequence as multiset and permutation then works as follows:

1. Encode the multiset $\mathcal{M}$, conditional on N and D (e.g. using the multiset coder from section 5.5.1).

2. Encode the permutation $x_1 \ldots x_N$ conditional on $\mathcal{M}$ (e.g. using the permutation coder from section 5.2).

Both approaches compress the original sequence down to exactly the same number of bits; they are mathematically equivalent. However, they are not computationally equivalent. The first approach, encoding the sequence element by element, is computationally faster, and also easier to implement.

Sequences with known occurrence counts

In the special case that the occurrence counts $\mathcal{M}$ of the sequence $x_1 \ldots x_N$ are known in advance to both the encoder and the decoder, only the permutation of the sequence has to be transmitted. The occurrence counts form a multiset as defined in equation (5.49), and implicitly determine the sequence length, because $|\mathcal{M}| = N$.

5.7.2 Sequences of known distribution, uncertain length

If the sequence length N is not known in advance, it has to be transmitted. Assuming that sequence lengths are distributed according to a distribution L, the probability mass of the sequence including its length is:

$$\Pr(x_1 \ldots x_N \mid D, L) \;=\; L(N) \cdot \prod_{n=1}^{N} D(x_n) \tag{5.50}$$

Here are two equivalent methods for encoding both sequence length and content:

1. **Explicit method.** Encode N first, using a code for L; then encode $x_1 \ldots x_N$ using one of the approaches of the previous section.

2. **Implicit method,** using an end-of-file (`EOF`) symbol. Encode, at each position n in the sequence, whether the sequence has more symbols or has terminated; and if it continues, the element x_n. The procedure stops when the end of the sequence is reached.

   Let $z_1 \ldots z_{N+1}$ be indicator variables $\in \{0, 1\}$ where $z_n = 1$ means that the sequence has a symbol at position n, and $z_n = 0$ that there are no more symbols (i.e. $N = n{-}1$). If the sequence length is already known to be longer than $n-1$, the probability distribution of the next indicator z_{n+1} is:

$$\Pr(z_{n+1} \mid L,\; N \geq n) \;=\; \left(1 - \frac{L(n)}{\sum_{k \geq n} L(k)}\right)^{z_{n+1}} \left(\frac{L(n)}{\sum_{k \geq n} L(k)}\right)^{1 - z_{n+1}} \tag{5.51}$$

$$\;=\; \text{Bernoulli}\Big(z_{n+1} \;\Big|\; L_{\setminus\{k \mid k<n\}}(n)\Big) \tag{5.52}$$

   Observe that $\Pr(z_1 \ldots z_{N+1}) = L(N)$. A suitable algorithm for this coding scheme sends, for each position n in the sequence, z_n followed by x_n. Each z_n is encoded using a Bernoulli code matching equation (5.52), and each x_n is encoded using a code matching D. After the last element x_N has been encoded, the termination marker $z_{N+1} = 0$ is sent.

The explicit method and the implicit method are mathematically equivalent and compress the sequence and its length down to exactly as many bits as given by the negative $\log_2$ of equation (5.50).

`EOF` markers are widely used in data compression algorithms, and an example of an implicit method. Many compression algorithms, rather than modelling L explicitly, treat `EOF` like an ordinary symbol of the alphabet, giving the sequence length an implicit probability distribution matching the occurrence of a yet unseen symbol. Compressors that employ `EOF` symbols in such a way include most variants of the PPM algorithm (described in chapter 6).

5.7.3 Sequences of uncertain distribution

Suppose a sequence $x_1 \ldots x_N$ is drawn *iid* from a discrete distribution D over some space $\mathcal{X}$, but the D itself is unknown. Various approaches to compressing such a sequence were discussed in chapter 4. Using the method that was described in section 4.4.1, we can compress the sequence while adaptively learning D by making Bayesian updates to a Dirichlet prior using the occurrences of each symbol in the sequence.[4] Integrating out D, the following compact form results:

$$\Pr(x_{n+1} = x \mid x_1 \ldots x_n,\ \boldsymbol{\alpha}) \ = \ \frac{\alpha_x + \mathcal{M}_n(x)}{A + n}, \tag{5.53}$$

where $\boldsymbol{\alpha}$ is the $|\mathcal{X}|$-length parameter vector of the Dirichlet prior, A is the sum of its components α_x, and $\mathcal{M}_n(x) \stackrel{\text{def}}{=} \#[x_k = x]_{k=1}^{n}$ counts how many times any given symbol x occurred in the sequence $x_1 \ldots x_n$.

Each x_n can therefore be encoded sequentially, updating the occurrence counts n_x after each symbol. For convenient choices of $\boldsymbol{\alpha}$, e.g. $\alpha_x = 1$ for all $x \in \mathcal{X}$, interfacing to an arithmetic coder becomes especially simple. To encode the length N of the sequence, the techniques from section 5.7.2 can be used just as before.

As was shown in section 4.4.3, an alternative to the natural sequential encoding of the sequence is to encode its symbol histogram first (e.g. using the multiset code from section 5.5.3), followed by its permutation (e.g. with the permutation code from section 5.2.1).

5.7.4 Sequences with known ordering

A sorted sequence $x_{(1)} \ldots x_{(N)}$ is a sequence whose elements have been arranged in a predefined order, where the ordering is known to both the sender and receiver. Sorting a sequence eliminates some of its degrees of freedom, which can be taken advantage of when compressing it. Specifically, a sorted sequence contains exactly as much information as the multiset of its symbol occurrences, as only the permutation information was destroyed by the sorting operation.

To encode a sorted sequence is therefore as simple or difficult as encoding a multiset, and any of the techniques from section 5.5 can be used.

An alternative to encoding the sorted sequence as a multiset is to encode it element by element, carefully transforming the probability distribution over the next element to take into account that the sequence is in sorted order. This transformation makes nearby elements more likely than far away elements, and gives zero mass to elements that would violate the ordering;

[4] It should be noted that the Dirichlet distribution, though mathematically convenient, is by no means the only possible choice of prior on D, and it may not always be appropriate.

however, the computational overhead of this method makes it somewhat unattractive for practical deployment.

5.8 Sets

5.8.1 Uniform sets

Suppose we want to encode a set $\mathcal{S} \subseteq \mathcal{X}$, where $\mathcal{X}$ is some known, finite superset. There are exactly $2^{|\mathcal{X}|}$ subsets of $\mathcal{X}$, so a uniform distribution over such sets would allocate probability mass

$$\Pr(\mathcal{S} \mid \mathcal{X}) = \frac{1}{2^{|\mathcal{X}|}} \tag{5.54}$$

to each set $\mathcal{S}$. Then we can encode any set $\mathcal{S}$ by a sequence of $|\mathcal{X}|$ binary digits $s_1 \ldots s_{|\mathcal{X}|}$, one for each element in $\mathcal{X}$, indicating whether that element appears in $\mathcal{S}$. The decoder, knowing $\mathcal{X}$ and the element ordering the encoder used, can reconstruct $\mathcal{S}$ from the sequence of digits. This coding procedure is optimal for uniformly distributed sets $\mathcal{S}$, as each element $x \in \mathcal{X}$ has an independent occurrence probability of $\frac{1}{2}$. The expected number of elements in a uniformly distributed set is $\mathbb{E}[|\mathcal{S}|] = \frac{1}{2}|\mathcal{X}|$.

5.8.2 Bernoulli sets

Generalising the above construction, suppose a set $\mathcal{S} \subseteq \mathcal{X}$ is generated by flipping a coin for each element of $\mathcal{X}$, and each element has an independent probability θ_x of being selected:

$$\Pr(x \in \mathcal{S}) \;=\; \theta_x \qquad \text{and} \qquad \Pr(x \notin \mathcal{S}) \;=\; 1 - \theta_x \tag{5.55}$$

The distribution over $\mathcal{S}$ is:

$$\Pr(\mathcal{S} \mid \mathcal{X}, \boldsymbol{\theta}) = \prod_{x \in \mathcal{X}} (\theta_x)^{\mathbb{1}[x \in \mathcal{S}]} \; (1 - \theta_x)^{\mathbb{1}[x \notin \mathcal{S}]} \,. \tag{5.56}$$

Such a set $\mathcal{S}$ can be encoded with an arithmetic coder by sequentially encoding the set membership for each element $x \in \mathcal{X}$ using a Bernoulli code of bias θ_x (see section 3.4.1).

5.9 Summary

This chapter introduced techniques for structural compression of various fundamental combinatorial objects, for various simple generative processes that produce such objects. Any compressor necessarily makes assumptions about the distribution of its input objects. Stating the assumed generative process of the inputs makes it easier to understand the properties

of a compression method, and helps in designing more complex models and appropriate compression algorithms.

Many of the objects in this chapter (e.g. sets, multisets, distributions) can also be represented as probability measures μ. For example, for a given set or multiset $\mathcal{M}$, define:

$$\mu = \frac{1}{|\mathcal{M}|} \sum_{x \in \mathcal{M}} \mathcal{M}(x) \delta_x \tag{5.57}$$

where δ_x is a Dirac delta function (point mass) located at x. Methods for drawing *iid* samples from such a probability measure μ (with or without replacement, and keeping or forgetting the order of the draws) correspond to natural generative processes for permutations, combinations, multisets and sequences. These relationships are summarised in Table 5.1.

Generative Process	**Input**	**Structure**	
draw without replacement	$\mathcal{M}$	permutation	combination
draw with replacement	D	sequence	multiset
draw with elimination	D	ordering	set
split an integer	N	composition	integer partition*
split a multiset	$\mathcal{M}$	ordered partition	partition*
		ordered	**unordered**

Table 5.1: Generative processes and resulting combinatorial structures.

*Not covered in this thesis.

Chapter 6

Context-sensitive sequence compression

This chapter reviews a family of sequence compression algorithms that model sequences one symbol at a time, using separate adaptive symbol distributions that depend on the *context* of the symbol. The context of a symbol is the sequence of symbols immediately preceding it, or a suffix thereof.

Notable examples of algorithms in this family include the "prediction by partial string matching" (PPM) algorithm by Cleary and Witten (1984a); its unbounded-depth variants PPM* (Cleary et al., 1995; Bunton, 1997); and compressors based on the probabilistic model named "Sequence Memoizer" (Wood et al., 2009), such as "Deplump" (Gasthaus et al., 2010; Bartlett and Wood, 2011).

This chapter makes several contributions to the understanding of this family of compression methods:

- Section 6.2 gives a unifying view on the construction of the algorithms in this family, showing the relationships between the existing algorithms.
- Sections 6.3 and 6.4 give an explicit form for the probabilistic model used in the *classic PPM algorithm* (Cleary and Witten, 1984a; Moffat, 1990; Howard, 1993), generalising its escape mechanism, and showing optimal settings of its parameters for various standard corpus texts.
- Section 6.5 makes explicit the probabilistic model of *PPM with blending*, similar to variants considered by e.g. Bunton (1997),[1] and shows its optimal parameter settings for standard corpus texts.

[1] Somewhat confusingly, Bunton uses different terminology in her papers: what she calls 'blending' is actually PPM's standard escape mechanism, and she refers to our notion of blending as 'mixing'.

- The influence of context depth on compression effectiveness is examined throughout sections 6.3–6.6. Data from deep or unbounded contexts can help to compress some data more effectively, but only in carefully controlled circumstances, and even then there are limited returns.
- Section 6.6 reveals the source of Deplump's compression effectiveness: its strength (at least for the files in the standard corpora) stems mainly from the use of depth-dependent parameters and parameter optimisation, rather than the use of unbounded-depth contexts.

6.1 Introduction to context models

The compression algorithms in this chapter compress sequences one symbol at a time; each symbol is compressed using a predictive probability distribution that depends on the preceding symbols. The operation of such an algorithm is much like a guessing game: at any point in the sequence, the algorithm tries to predict which symbol comes next, expressing its guess in the form of a probability distribution. This distribution is used to compress or decompress the next symbol using arithmetic coding. After each symbol, the algorithm updates its internal model to improve the guesses for the rest of the sequence.

Chapter 4 presented several adaptive models for sequences whose symbols are identically and independently distributed. This independence assumption is not appropriate for sequences of human text; for example, the letters `E`, `T`, `A`, `O`, `I`, `N` are the most frequent ones in English text; but when the preceding letter is `Q`, the most probable letter is `U`. More generally, symbols in human text are typically well predicted by the symbols that precede them.

A context-sensitive sequence compressor tries to take advantage of this property by using separate symbol distributions for each context. The *context* of a given symbol is the sequence of symbols immediately preceding it. For example, the length-3 context of symbol x_k is the sequence $(x_{k-3},\ x_{k-2},\ x_{k-1})$. Each context-dependent distribution is learned gradually from the data.

6.1.1 Pure bigram and trigram models

A *bigram model* selects a probability distribution for the next character based on the value of the previous character. There are therefore $|\mathcal{X}|$ separate symbol distributions in a bigram model, one for each symbol in the alphabet $\mathcal{X}$. These symbol distributions are typically learned from the data as compression proceeds. Consider a sequence whose first N symbols are observed, and suppose the last observed symbol x_N has value y. The bigram model selects

the probability distribution for the next symbol x_{N+1} based on its context of length 1 (i.e. the single symbol y). Let's call this distribution G_y.

Formally, the distribution induced by a bigram model can be written as follows:

$$\Pr(x_{N+1} = x \mid x_N = y,\ x_1 \ldots x_{N-1}) \ = \ G_y(x \mid \mathcal{M}_y) \tag{6.1}$$

where $\mathcal{M}_y$ is a sufficient statistic for G_y. For example, $\mathcal{M}_y$ can be set to the multiset of symbols that were immediately preceded by y in the sequence $x_1 \ldots x_N$:

$$\mathcal{M}_y \ := \ \biguplus_{n=2}^{N} \{\, x_n \mid x_{n-1} = y \,\} \tag{6.2}$$

Intuitively, $\mathcal{M}_y$ then counts how often, for each symbol x in the alphabet, the bigram (y, x) occurs in $x_1 \ldots x_N$. While this particular assignment of $\mathcal{M}_y$ is typical in bigram models, the $\mathcal{M}_y$ could in principle be set differently.

More generally, an n-gram model predicts the symbols in the sequence using contexts of length $n-1$, with a separate adaptive distribution $G_{\mathbf{s}}$ for each possible context $\mathbf{s}$. The next symbol x_{N+1} is predicted using the distribution selected by its context $\mathbf{s} = (x_{N-n+1} \ldots x_N)$:

$$\Pr(x_{N+1} = x \mid x_1 \ldots x_N) \ = \ G_{\mathbf{s}}(x \mid \mathcal{M}_{\mathbf{s}}) \tag{6.3}$$

where $\mathcal{M}_{\mathbf{s}}$ is a sufficient statistic for $G_{\mathbf{s}}$. In n-gram models, $\mathcal{M}_{\mathbf{s}}$ is commonly set to the multiset of those symbols in $x_1 \ldots x_N$ that were directly preceded by $\mathbf{s}$. For an alphabet of $|\mathcal{X}|$ symbols, an n-gram model could end up having to learn up to $|\mathcal{X}|^{n-1}$ separate symbol distributions. However, for $n \geq 3$ and sequences made from human text, this worst-case situation does not usually occur in practice.

Two difficulties arise when a pure n-gram model is to be used for sequence modelling or compression: Firstly, an n-gram model does not assign probability to the first $n-1$ symbols in the sequence, requiring a mechanism to deal with this special case. Secondly, the distribution for each context is learned separately from those of other contexts, even though there might be similarities between some of these contexts. (For example, the distribution of symbols following the subsequence `RTIO` might be similar to the distribution of symbols following `NTIO`.) A solution to both problems is to use probability distributions that combine information from contexts of different depths.

6.1.2 Hierarchical context models

In principle, any probability distribution over sequences can be factorised as follows:

$$\Pr(x_1 \ldots x_N) = \prod_{n=1}^{N} \Pr(x_n \mid x_1 \ldots x_{n-1}) = \prod_{n=1}^{N} G_{x_1 \ldots x_{n-1}}(x_n) \tag{6.4}$$

where each $G_{x_1 \ldots x_{n-1}}$ is a separate context-dependent distribution, selected by the preceding sequence $x_1 \ldots x_{n-1}$. Different context models correspond to different ways of assigning probability distributions to the Gs. (From here onwards, I will use the notation $x_{1:k}$ in subscripts as an abbreviation for $x_1 \ldots x_k$.)

Typically, various constraints are placed on these context distributions. For example, the family of bigram models is characterised by the constraint $G_{x_{1:N-1}} = G_{x_{N-1}}$, i.e. only the last symbol is used to select the distribution over the next symbol. Similarly, the constraints of an n-gram model can be described by $G_{x_{1:N-1}} = G_{x_{(N-n+1):(N-1)}}$.

A common assumption is that the distribution of a context $\mathbf{s}$ is similar to distributions of contexts that share a common suffix with $\mathbf{s}$. Let's write $\mathrm{suf}(\mathbf{s})$ for the longest proper suffix of $\mathbf{s}$:

$$\mathrm{suf}(x_{k:n}) := \begin{cases} x_{k+1:n} & \text{if } k < n \\ \varepsilon & \text{otherwise} \end{cases} \tag{6.5}$$

where ε is the empty sequence, and $\mathrm{suf}(\varepsilon)$ is undefined. For example, $\mathrm{suf}(\texttt{ABC}) = \texttt{BC}$, $\mathrm{suf}(\texttt{BC}) = \texttt{C}$ and $\mathrm{suf}(\texttt{C}) = \varepsilon$. Our expectation is that $G_{\mathbf{s}}$ is similar to $G_{\mathrm{suf}(\mathbf{s})}$.

One way of incorporating this sharing of information is to define the distributions *recursively*, e.g. by making each distribution $G_{\mathbf{s}}$ a function of the distribution $G_{\mathrm{suf}(\mathbf{s})}$. This recursive construction is one of the characteristic features of the sequence compression algorithms covered in this chapter. Rather than being fixed distributions, all the $G_{\mathbf{s}}$ are learned adaptively from the data.

As an example of such a recursive model, consider the following definition for $G_{\mathbf{s}}$:

$$G_{\mathbf{s}}(x_{N+1}{=}x \mid \mathcal{M}_{\mathbf{s}}) := \frac{\mathcal{M}_{\mathbf{s}}(x)}{|\mathcal{M}_{\mathbf{s}}| + \alpha} + \frac{\alpha}{|\mathcal{M}_{\mathbf{s}}| + \alpha} \cdot \begin{cases} \dfrac{1}{|\mathcal{X}|} & \text{if } \mathbf{s} = \varepsilon \\ G_{\mathrm{suf}(\mathbf{s})}(x) & \text{otherwise} \end{cases} \tag{6.6}$$

where α is a strength parameter, $\mathcal{M}_{\mathbf{s}}$ is a multiset of symbol occurrences for context $\mathbf{s}$, and $|\mathcal{X}|$ is the size of the alphabet. The distributions $G_{\mathbf{s}}$ and summary multisets $\mathcal{M}_{\mathbf{s}}$ are adaptively constructed and therefore depend on the preceding sequence $x_1 \ldots x_N$, but for ease of notation this dependency is quietly omitted here.

Note how equation (6.6) corresponds closely to the adaptive method from section 4.2.2. The way the summary multisets $\mathcal{M}_{\mathbf{s}}$ are chosen has a profound effect on the probability distri-

butions G_s. A common choice for hierarchical models is the *shallow update rule*, which is described in section 6.3.3. The distribution (6.6) can be interpreted as a sequential construction of a draw from a hierarchical Dirichlet process (with a uniform base distribution), if the $\mathcal{M}_s$ are set and updated correctly. Details are described by e.g. Teh et al. (2006).

6.2 General construction

The compression algorithms discussed in this chapter have the following functionally separable components:

- An algorithmic *engine* that comprises the mechanisms for computing, memorising and providing access to summary information about each context. Engines may have technical restrictions on the kinds of models that can be used; e.g. finite versus infinite context depth.
- A (hierarchical) *probabilistic model*, which specifies the way symbol probabilities are calculated from the summary information. The probabilistic models of many classic PPM variants are defined implicitly by specifying a 'smoothing method' and an 'update rule'.
- A *learning mechanism*, which updates the summary information in a way that allows the model to improve its predictions (e.g. exact Bayesian inference, an approximate method, or an ad-hoc method).
- An *arithmetic coder*, and a *region mapping mechanism* for the probabilistic model. In conjunction, these two components are responsible for the encoding or decoding of the compressed output sequence.

These components may be chosen more or less independently of each other, but they have to be well matched to produce sensible results.

Most of the classic PPM variants are specified through a *smoothing method* and an *update rule*, which (in conjunction) implicitly define a probabilistic model. The compressors based on the Sequence Memoizer, on the other hand, state the probabilistic model explicitly, and derive matching approximate Bayesian learning algorithms. (These algorithms could in turn be interpreted as smoothing methods and associated update rules.)

One parameter common to most of these models is the maximum context depth D, which can be finite or infinite. Some engines pose restrictions on the kind of models that can be used; for example, the engine of the classic PPM algorithm does not support unbounded depth,

and the engine of the "Deplump" compressor imposes certain restrictions on the probabilistic model (these are described in section 6.6.5).

An overview of different compression methods that are unified by this construction can be found in Table 6.1.

6.2.1 Engines

The engine includes the data structure and mechanism responsible for storing and updating the statistics for different contexts, and making them available to other parts of the algorithm. A commonly used data structure used by such an engine is the *search trie*, a tree that stores data (such as symbol counts) indexed by a context string. Here is a list of several existing engines:

- Classic PPM engine (Moffat, 1990), using fully-expanded context tries with vine pointers. This engine is limited practically to contexts of bounded depth. A description of this engine is given in section 6.3.
- Classic PPM* engine (Cleary et al., 1995) based on Patricia tries. This engine collapses non-branching nodes and uses back-pointers into the original string. The PPM* engine was the first to support contexts of unbounded depth.
- Finite state automaton engine (Ukkonen, 1995; Bunton, 1996). This engine carefully collapses non-branching nodes and temporarily re-expands them when needed, maintaining a set of "active nodes". Bunton used this engine to re-implement PPM*.
- Reverse suffix tree construction (Gasthaus et al., 2010) used by Deplump. For every symbol, the tree is traversed from the root (from the current symbol backwards through the observed string) to find the longest suffix. Non-branching nodes are collapsed into single nodes.

These engines are functionally equivalent (except for technical restrictions as mentioned above), and each engine will produce identical compression results for any of the probabilistic models it supports.

6.2.2 Probabilistic models

The models used in context-sensitive compression algorithms have the following common structure: they define, for any given context $\mathbf{s}$, an adaptively changing (conditional) probability distribution $G_{\mathbf{s}}$ over symbols $x \in \mathcal{X}$, using the summary information available for $\mathbf{s}$

Method	Depth	Smoothing	Updates	Details	Reference
PPM{A,B}	finite	backing off	full	in Table 6.2	Cleary and Witten (1984a)
PPMC	finite	backing off	shallow	in Table 6.2	Moffat (1990)
PPM{P,X}	finite	backing off	shallow	see reference	Witten and Bell (1991)
PPMD	finite	backing off	shallow	in Table 6.2	Howard and Vitter (1991, 1992)
PPME	finite	backing off	shallow	in Table 6.2	Åberg and Shtarkov (1997)
PPMG	finite	backing off	shallow	strength α, discount β	section 6.4.2
BPPM	finite	blending	shallow	strength α, discount β	section 6.5
PPM*	unbounded	backing off	full	PPMC + state selection	Cleary et al. (1995)
PPM* SB	unbounded	various	various	PPMD + state selection	Bunton (1997)
PPMII	finite	custom	custom	see references	Shkarin (2001a,b, 2002)
SeqMem	unbounded	blending	stochastic	depth-dep. (α, β)	Wood et al. (2009, 2011)
Deplump 1	unbounded	blending	various	depth-dep. (α, β)	Gasthaus et al. (2010)
Deplump 2	unbounded	blending	various	depth-dep. (α, β)	Bartlett and Wood (2011)

Table 6.1: Overview of various context-sensitive sequence compression methods. Many of the above are unified in this chapter. The probabilistic models of these compressors (except for Sequence Memoizer and Deplump), are specified implicitly through a smoothing method and update rule.

and the distribution $G_{\text{suf}(\mathbf{s})}$ of the next shorter context. Probabilistic models of this kind are called *hierarchical models*, and historically also known as *smoothing methods*.

An overview of smoothing methods and their empirical performance on word-based corpora is given by Chen and Goodman (1996, 1998). Rather than giving an exhaustive treatment of smoothing methods for compression here, I will distinguish two main categories of approaches:

(I) Switching to the shorter context when "not enough" data has been observed in the longer context. This family of methods is called *backing off*. The resulting probability distribution can be described by an urn scheme with an exclusion rule.

(II) Mixing the probability distributions for the longer and shorter contexts using a weighted sum of the component distributions. The weights are a function of the symbol observation counts. This approach is called *blending* (or 'interpolating').

Methods from category I are cheaper to compute, as the backing-off mechanism does not need to access the counts of all context depths for every symbol. Backing-off effectively ignores some of the accumulated information. Its advantages over the methods from category II are increased computation speed and implementation simplicity. The "probability estimation methods" used in the classic variants of the PPM algorithm belong to category I.

Methods from category II compute their predictions involving the symbol counts from all context depths, and can in principle give better predictions than methods from category I. Examples of methods from category II include hierarchical Dirichlet processes and hierarchical Pitman–Yor processes, and are used in sequence prediction models such as the language model in Dasher (Ward et al., 2000), or the Sequence Memoizer by Wood et al. (2009).

Historical notes. Smoothing methods have a fair amount of history, and some attempts have been made to summarise and compare them, e.g. by Chen and Goodman (1996, 1998) and by Champion (1997). In principle, any smoothing method could be used for compression, as long as it allocates non-zero probability mass to all symbols in the alphabet.
Chen and Goodman concluded in the technical report that their modification to Kneser–Ney smoothing (Kneser and Ney, 1995) performed best: blending rather than back-off, and separate parameters for different context depths. Bayesian treatment arrived with: Nádas (1984), hierarchical Dirichlet models (MacKay and Bauman Peto, 1995; Teh et al., 2006), hierarchical Pitman–Yor models (Teh, 2006a,b).

Context depth. While increasing the context depth leads to more precise predictions, each context appears less frequently, leading to sparse data and potentially less accurate predictions (and thus worse compression). In the extreme case where a context appears for the first time and no symbol has ever been observed in that context, this problem is called the "zero frequency problem" (Witten and Bell, 1991; Cleary and Teahan, 1995). But more than just a zero frequency problem, it's really a general sparsity problem.

6.2.3 Learning

Context-sensitive sequence compressors typically maintain a collection of sufficient statistics that summarise their knowledge about the symbol distributions of different contexts. The algorithm attempts to learn these distributions by updating its sufficient statistics as it sees more of the sequence.

It is important to note that probabilistic model and learning algorithm are intimately connected, and cannot be considered independent components. In a fully Bayesian treatment, all assumptions are specified by the model, and the correct learning method can be derived mechanically. However, an exact algorithm for the 'correct' learning method does not exist for all models. Two approaches might be followed in that case:

- using an approximate learning algorithm (such as MCMC methods or particle filtering).
- changing the model, or using an 'incorrect' method that still works reasonably well.

One might argue that these two approaches are not entirely disjoint concepts: the use of some approximate learning algorithms might be considered changing of the model. For example, Deplump's UKN-inference algorithm is an approximate inference method for the hierarchical Pitman–Yor process, but the algorithm could also be understood as defining a different probabilistic model.[2]

The early variants of PPM algorithm employed one of two popular learning mechanisms, named 'full updates' and 'shallow updates' in section 6.3.3. These learning methods were not

[2]The difference between these two probabilistic models is made clear in equations (6.14) and (6.16).

derived in a Bayesian way, but discovered empirically (Moffat, 1990, PPMC) or chosen for algorithmic convenience (Cleary and Teahan, 1995, PPM*).

The algorithms based on the Sequence Memoizer primarily use approximate Bayesian learning methods that are designed not to bias the probabilistic model. A practical approximate learning algorithm may need to trade off among accuracy, computational resource demands, and perhaps ease of implementation.

Overview

Having introduced this general family of compression algorithms, some specific constructions are now discussed in more detail. Section 6.3 describes the PPM algorithm of Cleary and Witten (1984a) with the modifications by Moffat (1990) and Howard (1993), and briefly discusses its relations to other PPM variants. Section 6.4.2 introduces PPMG, a two-parameter generalisation of PPM's escape mechanism, and reports the optimal setting of its parameters (α,β) for various standard corpus files. Section 6.5 modifies PPM's generalised escape mechanism from back-off with blending, yielding a model similar to the "interpolated Kneser–Ney" procedure described by Chen and Goodman (1998). This algorithm variant, named 'BPPM', is then optimised along the same lines as in the previous section.

Finally, unbounded-depth variants of these algorithms are considered in section 6.6.

6.3 The PPM algorithm

The "prediction by partial string matching" (PPM) algorithm is a compression method that combines predictions from different contexts using backing-off. It processes an input sequence symbol by symbol, accumulating counts of symbol occurrences for different contexts (up to some maximum depth D), and using those counts to compress the input with an arithmetic coder.

6.3.1 Basic operation

Data structure. The classic PPM algorithm makes use of a data structure known as a *trie*, a search tree that maps partial strings (contexts) up to some maximum length D to a histogram of symbol occurrences. Each node stores one histogram, and typically also a pointer to the node of the next shorter context; these pointers are called *vine pointers*. The vine pointers help with two things: firstly, they allow the algorithm to retrieve the next shorter context quickly when computing the predictive symbol distribution; secondly, they speed up finding the context node for the next symbol in the sequence, when the finite-depth context window

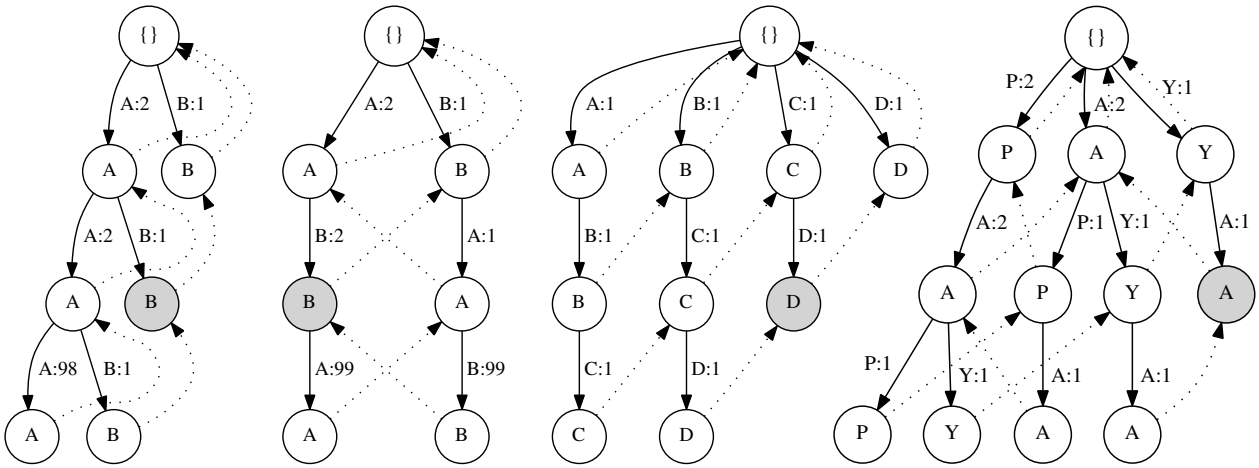


Figure 6.1: Diagrammatic representations of some basic PPM trie structures of depth 3. Each node uniquely identifies a context $\mathbf{s}$, and the labels of its outgoing arrows are the components of its summary multiset $\mathcal{M}_{\mathbf{s}}$. Vine pointers are shown with dotted lines. The shaded node indicates the node pointer after processing the last symbol. From left to right: (6.1a) The string consisting of 100 `A`s, followed by one `B`. (6.1b) The string `AB` repeated 100 times. (6.1c) The string `ABCD`. (6.1d) The string `PAPAYA`.

slides ahead by one symbol. Graphical examples of trie structures for simple strings are shown in Figure 6.1.

Algorithm. In its basic form, the PPM algorithm works as follows:

1. Initialise the search trie with an empty root node.
2. Repeat until stop criterion is reached:
   (a) **Fetch.** Retrieve the next input symbol x.
   (b) **Encode.** Use the probability distribution defined by the histogram of the current trie node (and its parents) to encode x. See section 6.3.2 for details.
   (c) **Learn.** Update the histogram of the current trie node, and also the histograms of the nodes along the vine pointer chain. See section 6.3.3 for details.
   (d) **Advance.** Find (or create) the node in the trie corresponding to the next context, creating any missing nodes. Let `p` be a pointer to the current node.
      i. If the current node is at depth D in the trie, set `p` to the vine pointer.
      ii. If the node pointed to by `p` does not have a child labelled x, create one. (Make sure to set the new child's vine pointer correctly, possibly by creating additional nodes.)
      iii. Point `p` to the child labelled x. The node identified by `p` is the new current node.

The stop criterion of the loop in step 2 may be that the end of the input sequence has been reached, or that a predetermined number of symbols have been processed.

The beauty of this adaptive procedure, common to all sequence compressors of this kind, is that the decoder can construct the same data structure and histograms when decompressing as the encoder did during compression. Although the predictive probability distribution changes adaptively after every step, each symbol is encoded and decoded using only information obtained from the previous symbols — this information is available to both the encoder and the decoder.

The part of the algorithm that most affects compression effectiveness is the probabilistic model, which defines how symbols are encoded with the adaptively learned histograms, and how the histograms are updated to incorporate new symbols.

Most developments after PPM's initial publication were modifications to one or both of these two components. A description of the original "probability estimation methods" and update rules will follow shortly, along with a review of the history of PPM.

6.3.2 Probability estimation

All PPM algorithms use arithmetic coding to compress the sequence of input symbols, whose symbol probabilities are computed from the adaptively changing histograms. PPM traditionally assigns those probabilities using an "escape mechanism", a computationally convenient method which avoids making excessive lookups from the trie. The original form of the PPM algorithm as published by Cleary and Witten (1984a) suggested two such escape mechanisms (A and B), but better methods exist; these will be reviewed and generalised in section 6.4.2. For simplicity of presentation, PPM's escape mechanism will be described using method A (which somewhat resembles the Laplace rule from section 4.2.1).

Given the symbol histogram of the current context, let N denote the total number of symbols observed, and n_x the number of times symbol x was seen. If $n_x > 0$, symbol x is coded with probability mass proportional to n_x; otherwise, a special *escape symbol* (`ESC`) is coded to communicate that the symbol has not yet occurred in this context.[3] Method A defines these probabilities as follows:

$$\Pr(x) = \frac{n_x}{N+1} \qquad \text{and} \qquad \Pr(\texttt{ESC}) = \frac{1}{N+1} \tag{6.7}$$

Whenever an escape event is coded, the symbol x itself still remains to be communicated. To do this, the coding procedure is repeated recursively with the next shorter context (which is easily accessible from the vine pointer). Eventually a context is reached in which the symbol

[3] `ESC` (somewhat like `EOF`) is a virtual symbol that does not occur in the input sequence.

was seen before, and at that point the recursion terminates. If x has never been seen in any context, the root node of the trie is ultimately reached and an escape event is coded there, too. When that happens, x is finally encoded with a uniform distribution over all possible symbols.

The approach as outlined so far works fine, but it does not use its permissible coding range optimally, wasting information bandwidth. To see this, consider a context in which exactly one symbol (`X`) has been observed previously, and now a new symbol (`Y`) is encountered. The algorithm codes `ESC` and backs off to the next shorter context, where `X` has also been seen before. But after processing `ESC` the decoder already knows that the next symbol cannot be `X`, or any of the symbols observed in the previous context, because the encoder chose `ESC` rather than an observed symbol.

This problem can be resolved by excluding from the shorter contexts all symbols that were observed at least once in the longer contexts. By treating the counts of excluded symbols as zero counts, the probability mass of excluded symbols is redistributed to the remaining symbols in the context. The algorithm can communicate those symbols by passing up a set of excluded symbols when it backs off to the next shorter context. Alternatively, it can just pass the pointer of the previous context node, as the set of observed nodes in any given context is always a subset of the set of observed nodes in the next shorter context.

Writing $\mathcal{R}$ for the set of excluded symbols and $R = \sum_{x \in \mathcal{R}} n_x$ for the total number of excluded symbol occurrences, PPM's probability assignment under escape method A and the exclusion rule becomes:

$$\Pr(\texttt{ESC} \mid \mathcal{R}) = \frac{1}{N - R + 1} \qquad \text{and} \qquad \Pr(x \mid \mathcal{R}) = \begin{cases} 0 & \text{for } x \in \mathcal{R} \\ \dfrac{n_x}{N - R + 1} & \text{otherwise} \end{cases} \tag{6.8}$$

Cleary and Witten (1984a) documented equation (6.7) in their paper, but used equation (6.8) for their results. This symbol-exclusion modification is well known, but rarely stated explicitly. In the PPM literature, the assignment of symbol and escape probabilities are typically stated in plain form, without taking symbol exclusion into account.

To be pedantic, the algorithm as currently stated has one remaining information leak. Consider a context in which all symbols of the alphabet have already been seen at least once. Then the coding method as specified by equation (6.8) still allocates mass to `ESC`, even though `ESC` can no longer occur. So strictly speaking, the `ESC` symbol should not get any probability mass in this situation. As far as I am aware, none of the existing PPM implementations attempt to close this gap; and in practice it's probably not worth the effort, as the amount of information lost through this situation is at worst $\log_2 \frac{|\mathcal{X}|+1}{|\mathcal{X}|}$ bits per symbol, and diminishes in the limit as more symbols are processed.

6.3.3 Learning mechanism

Every time PPM has processed a new symbol, it updates the histograms in its trie data structure to reflect the new observation. This is how PPM learns the characteristics of the input sequence, allowing it to improve its predictive probability distributions for future symbols. The update method used in the original version of PPM by Cleary and Witten (1984a) can be stated as follows:

> **Full updates.** After observing a symbol x, update its count in the current context, and in all shorter contexts (including the empty context). The counts produced by this rule correspond exactly to those of a full n-gram model.

This update rule seems nice and natural, as each node counts exactly how often each symbol occurs in its context. It turns out, however, that compression effectiveness (and efficiency) improves with the following modification, introduced by Moffat (1990):

> **Shallow updates.** After observing a symbol x, increment its count in the current context. If x was observed for the first time, recursively update the next shorter context using the same rule. [The counts produced by this rule correspond exactly to those produced by the "1 table per dish" (1TPD) approximation for hierarchical Pitman–Yor processes.]

In nearly all implementations, shallow updates are used. Shallow updates tend to give better results than full updates; see e.g. MacKay and Bauman Peto (1995) for insights why this is the case. (Shallow updates are sometimes named "update exclusions" in the PPM literature.)

6.3.4 PPM escape mechanisms

The PPM algorithm, as published by Cleary and Witten (1984a), proposed two escape methods, A and B, and the use of full updates and symbol exclusions. This form of the algorithm was refined later by Moffat (1990), adding vine pointers, shallow updates, and an escape method C. Escape methods were considered more generally by Witten and Bell (1991), also outside the PPM algorithm, adding Poisson-process-based methods P and X and comparing them to the existing methods. Howard (1993) proposed escape method D, which remains, to the best of my knowledge, the most effective escape method for the classic PPM algorithm on standard corpora. Method E was suggested by Åberg et al. (1997) as an improvement over method D, but it's not a clear winner; for details, see the compression results in appendix A, and the contour plots in Figures 6.4 and 6.5.

All the above escape methods differ only in the assignment of symbol and escape probabilities, and in the criterion used to determine when to trigger the escape. An overview of notable PPM

	PPMA	**PPMB**	**PPMC**	**PPMD**	**PPME**	**PPMG**
$\Pr(x)$	$\frac{n_x}{N+1}$	$\frac{n_x-1}{N}$	$\frac{n_x}{N+U}$	$\frac{2n_x-1}{2N}$	$\frac{4n_x-2}{4N-1}$	$\frac{n_x-\beta}{N+\alpha}$
$\Pr(\texttt{ESC})$	$\frac{1}{N+1}$	$\frac{U}{N}$	$\frac{U}{N+U}$	$\frac{U}{2N}$	$\frac{2U-1}{4N-1}$	$\frac{U\beta+\alpha}{N+\alpha}$
Escape if:	$n_x=0$	$n_x\leq 1$	$n_x\leq 1$	$n_x=0$	$n_x=0$	$n_x=0$

Table 6.2: Overview of classic PPM escape methods. The table shows each method's escape criterion, and its assignments to symbol and escape probabilities. N denotes how many symbols were seen in total (in the current context), n_x how many of those were the symbol x, and U the number of unique symbols (among those N).
PPMA and PPMB were published by Cleary and Witten (1984a). PPMC was suggested by Moffat (1990). PPMD was proposed by Howard (1993), and PPME by Åberg et al. (1997). A parametrised generalisation, PPMG, is described in section 6.4.2.

escape methods can be found in Table 6.2. Section 6.4.2 introduces method G, a continuous generalisation of escape methods A, D and E.

Section 6.5 describes variants of PPM that replace backing off with blending.

6.3.5 Other variants of PPM

Many modifications to the PPM algorithm have been proposed in the literature, often improving compression effectiveness at the expense of significantly complicating the original algorithm. Some of those modifications seem elaborate hacks which do not necessarily offer much insight into why they work. These modifications are not considered further in this chapter. A good summary of useful innovations to the PPM algorithm between 1984 and 2005 is given by Korodi and Tabus (2008).

The PPM algorithm can be amended to support contexts of unbounded depth. Cleary et al. (1995) proposed such a modification, named PPM*, which includes a new trie data structure that permits collapsing non-branching paths into single nodes. Unbounded depth variants of PPM are discussed in section 6.6.

Shkarin (2001a) published a finite-depth algorithm named PPMII (PPM with information inheritance), which uses custom probability estimation and update rules that were manually optimised for good compression and speedy computation. As of 2014, the results produced by PPMII are still state of the art and unbeaten by any other known PPM variant. PPMII was included as a compression method in the ZIP file format by Peterson et al. (2006).

A PPM variant not discussed here is PPMZ by Bloom (1998). There are some independently developed compression algorithms which were later shown to relate to PPM. These include

DMC (Cormack and Horspool, 1987), whose relationship to PPM is shown by Bunton (1996); and perhaps CTW by Willems et al. (1993, 1995). Cleary et al. (1995) mention a connection between the Burrows–Wheeler transform and infinite context tries (as constructed by PPM*), which is investigated further by Fenwick (2007). None of these are discussed further in this chapter, but compression results for all the above methods are included in appendix A for comparison.

6.4 Optimising PPM

This section visualises the effects of the choice of D, the maximum context depth of the PPM algorithm. It also offers a slightly more general escape mechanism for classic PPM, and jointly optimises its parameters alongside D for some files of standard compression corpora.

6.4.1 The effects of context depth

The finite-depth PPM algorithm is parametrised by a maximal context depth D. One might expect that larger values of D allow capturing information from longer contexts, therefore improving the algorithm's predictive power and thus its compression effectiveness. Somewhat surprisingly, however, it turns out that the optimal setting for typical English language text files (of length up to 10^7 symbols) is at around $D = 5$. This effect is demonstrated in Figure 6.2: compression effectiveness initially increases with D until it peaks at $D = 5$, and deteriorates slightly beyond 6.

These results seem to justify that a setting of $D = 5$ could be hard-wired into the algorithm, and indeed this appears to be fairly common practice. But the optimal choice of D also depends on the input sequence and its length. Figure 6.3 demonstrates how, for a long text file, the optimum setting of D can change as a function of the file's length.

Conceptually, there's something dissatisfying about imposing a bound on the context depth in the first place: Surely, a sensible algorithm should become more effective at compressing as it gains deeper knowledge of its input sequence. It appears that the classic PPM algorithm simply isn't putting added depth to good use.

As it turns out, added context depth *can* be used effectively. Unbounded-depth variants of PPM and related algorithms are explored in section 6.6.

6.4.2 Generalisation of the PPM escape mechanism

Escape method G. This section presents a generalised escape method G, which parametrises the symbol and escape probabilities with two continuous parameters α and β. Method G de-

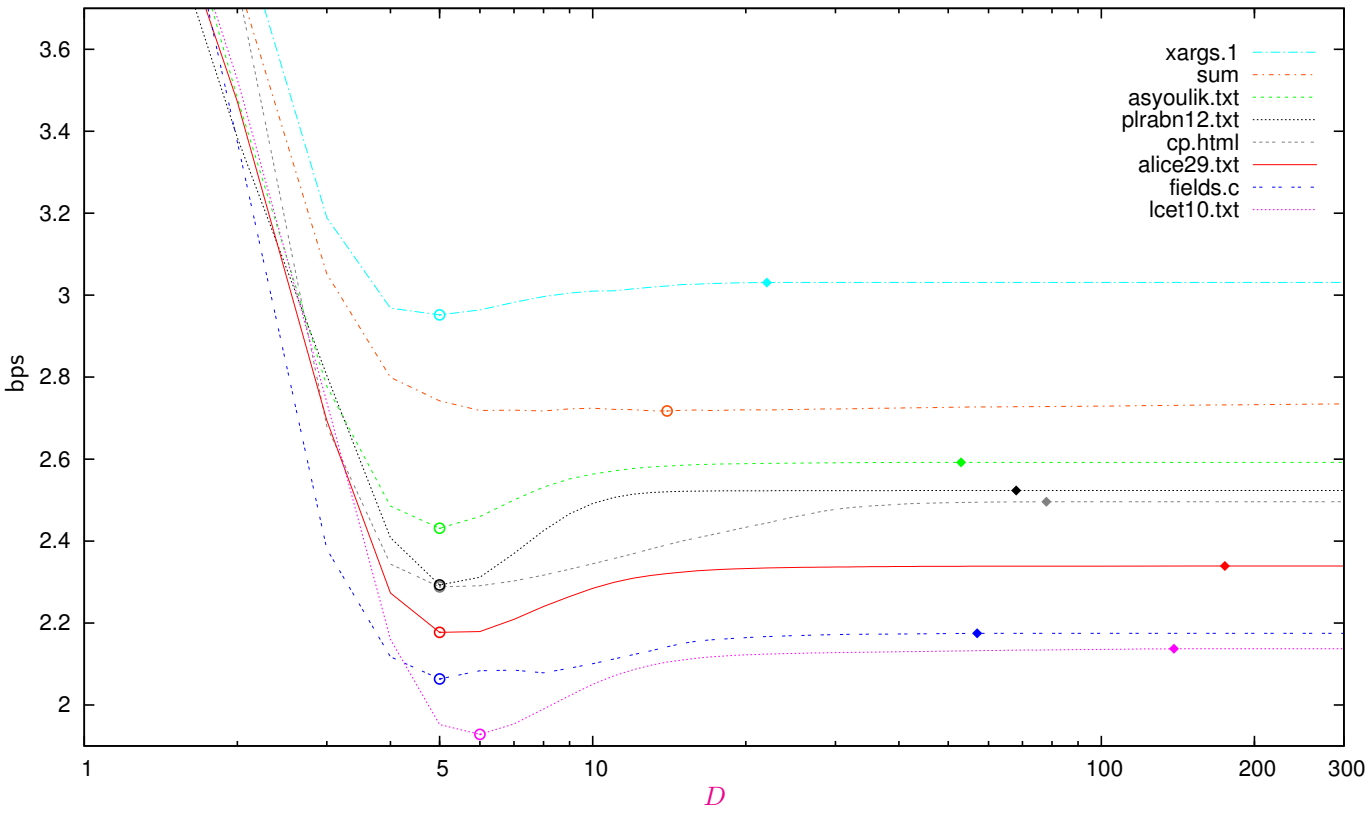


Figure 6.2: PPMD's compression effectiveness in bits per symbol, measured on various English text files of the Canterbury corpus, as a function of D (the maximum context depth). For each of the corpus files, the optimal choice of D is indicated by a circle. The solid diamonds mark the points beyond which additional depth has no further effect. Details about the Canterbury corpus can be found in appendix A.

fines the symbol and escape probabilities as follows:

$$\Pr(x) = \frac{n_x - \beta}{N + \alpha} \cdot \mathbb{1}[n_x > 0] \qquad \text{and} \qquad \Pr(\texttt{ESC}) = \frac{U\beta + \alpha}{N + \alpha}, \tag{6.9}$$

where N is the number of total symbol observations in this context, and U is the number of unique symbol observations. The parameter $\beta \in [0, 1]$ is a discount parameter, and $\alpha \in [-\beta, \infty)$ is a concentration parameter, similar to those in a Pitman–Yor process.

The explicit form of method G, taking into account symbol exclusions, is:

$$\Pr(\texttt{ESC} \mid \mathcal{R}) = \frac{U\beta + \alpha}{N - R + \alpha} \quad \text{and} \quad \Pr(x \mid \mathcal{R}) = \begin{cases} 0 & \text{for } x \in \mathcal{R} \\ \dfrac{n_x - \beta}{N - R + \alpha} \cdot \mathbb{1}[n_x > 0] & \text{otherwise,} \end{cases} \tag{6.10}$$

where $\mathcal{R}$ and R are defined as for equation (6.8). Escape method G generalises methods A, D and E: PPMA is recovered for ($\alpha = 1$, $\beta = 0$), PPMD for ($\alpha = 0$, $\beta = \frac{1}{2}$), and PPME for ($\alpha = -\frac{1}{4}$, $\beta = \frac{1}{2}$). As can be seen from Table 6.2, escape methods B and C are not generalised

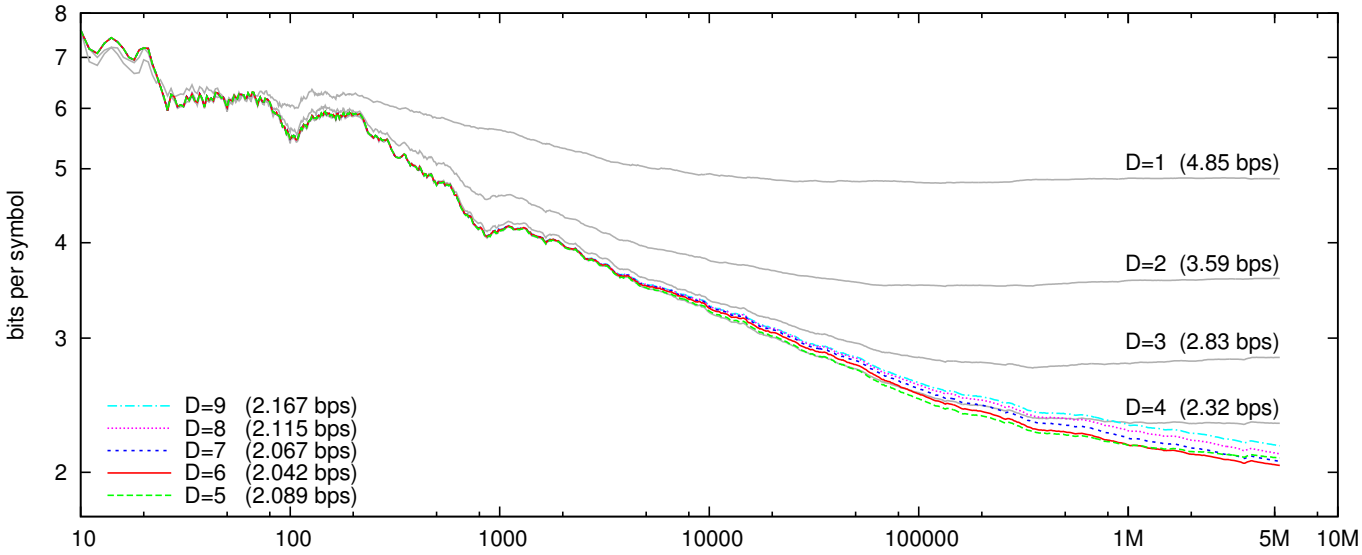


Figure 6.3: PPMD's compression effectiveness as a function of input length, for different settings of the maximum context depth D. The input sequence is `shakespeare.txt` (ca. 5 MB), Shakespeare's plays in concatenated plain text. The plot shows how the optimal setting of D varies as more of the sequence is processed. For compressing the entire file, $D = 6$ is the optimal setting. If only the first 100 000 bytes of the file are compressed, $D = 5$ is best.

by PPMG, because they use a different escape criterion.

Given that these escape methods can be viewed as locations in the 2-dimensional continuum spanned by α and β, one might wonder which location $(\alpha_{\mathsf{OPT}}, \beta_{\mathsf{OPT}})$ actually maximises PPM's compression effectiveness. The location of the optimum clearly depends on the sequence being compressed and on the choice of the depth D. Figure 6.4 contains a contour plot of PPMG's compression effectiveness as a function of α and β (for fixed input text and fixed D), showing the location of the optimum and the locations corresponding to escape methods A, D and E. Table 6.3 shows the settings of α, β, and D that jointly optimise PPM's compression effectiveness for several files of the Canterbury corpus.

File	α_{OPT}	β_{OPT}	D_{OPT}	compression rate (bps)
`alice29.txt`	0.223	0.366	5	2.1684
`asyoulik.txt`	0.290	0.380	5	2.4233
`fields.c`	−0.260	0.510	5	2.0360
`lcet10.txt`	0.063	0.398	6	1.9199
`plrabn12.txt`	0.390	0.300	5	2.2795
`xargs.1`	−0.085	0.522	5	2.9510
Canterbury (complete)	−0.040	0.554	5	1.7082

Table 6.3: Most effective possible compression with PPMG for various corpus files, and the approximate setting of the parameters $(\alpha_{\mathsf{OPT}}, \beta_{\mathsf{OPT}}, D_{\mathsf{OPT}})$ for which the optimal compression is achieved.

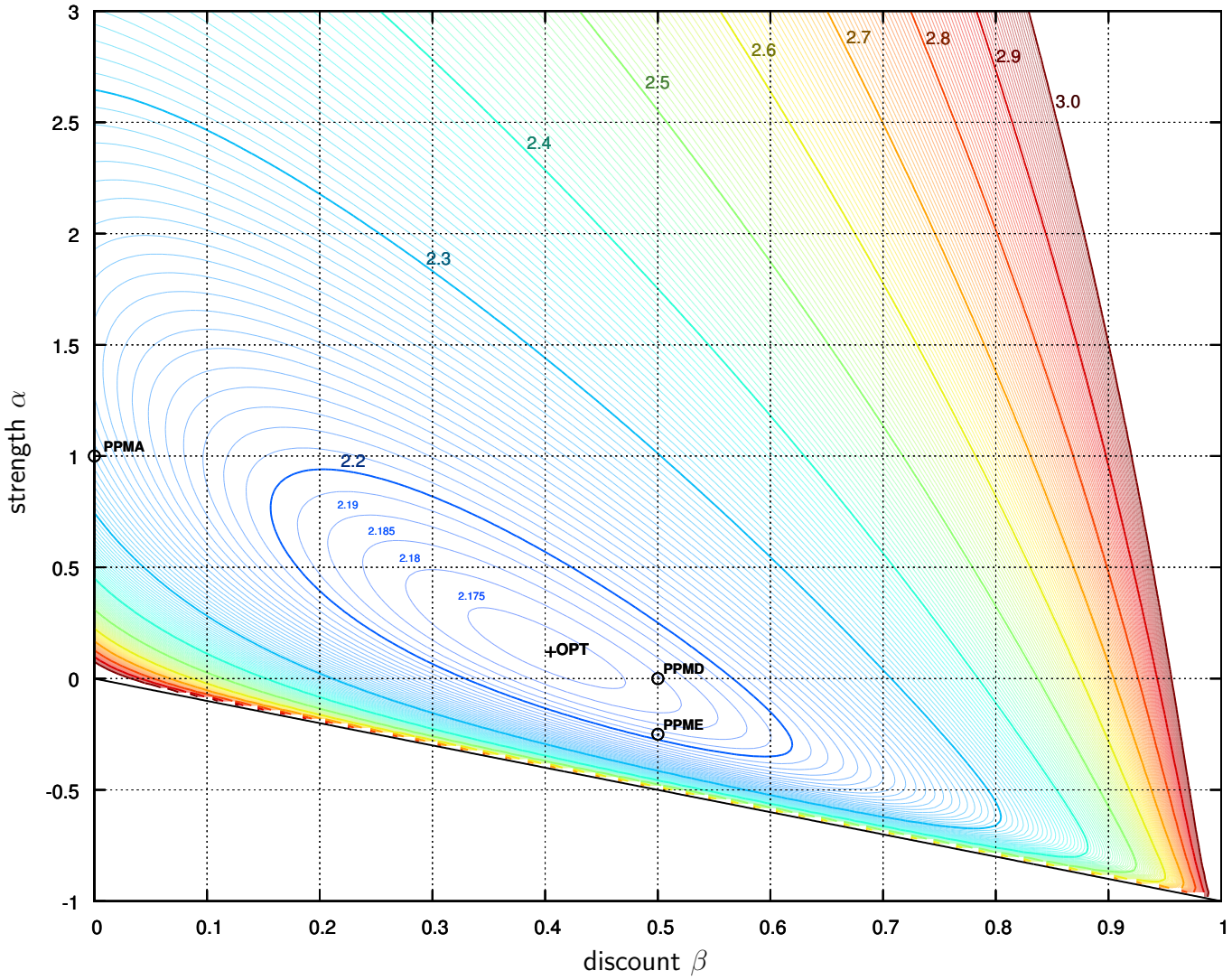


Figure 6.4: Contour plot of PPMG's compression effectiveness in bits/symbol, measured for `alice29.txt` as a function of strength parameter α and discount β, with maximum context depth $D = 6$. The optimum of 2.1726 bits per symbol is obtained for $\alpha \approx 0.115$ and $\beta \approx 0.405$. Traditional escape method PPMA corresponds to setting $\alpha = 1$ and $\beta = 0$ (yielding 2.264 bps), PPMD corresponds to $\alpha = 0$ and $\beta = \frac{1}{2}$ (yielding 2.179 bps), and PPME to $\alpha = -\frac{1}{4}$ and $\beta = \frac{1}{2}$ (yielding 2.194 bps). The results for other corpus files are similar, and summarised in Figure 6.5.

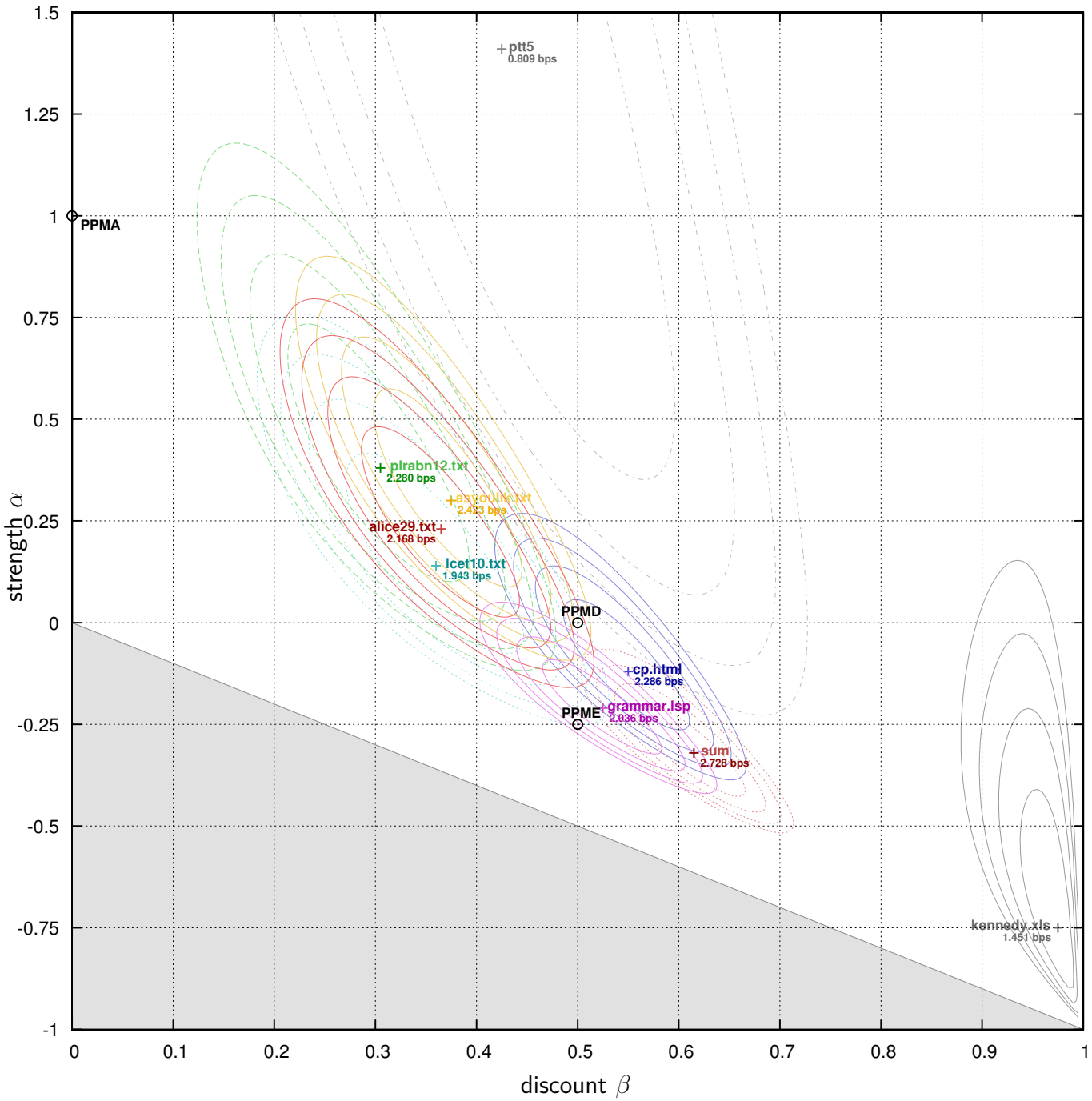


Figure 6.5: The optimal settings of the strength α and discount β parameters of PPMG with maximum context depth $D = 5$, for a variety of files from the Canterbury corpus. The optimal (α, β) for each file is indicated with a plus sign next to the file's name. The innermost contour marks a distance of 0.0025 bits per symbol from the optimum (indicated by a plus). Adjacent contours (of the same colour) are separated from each other by another 0.0025 bits per symbol.

6.4.3 The probabilistic model of PPM, stated concisely

The symbol distributions $G_{\mathbf{s}}$ of PPM's traditional escape mechanism use symbol multisets $\mathcal{M}_{\mathbf{s}}$ as sufficient statistics (for all contexts $\mathbf{s}$). A characteristic feature of these distributions is that symbols that were observed at least once in context $\mathbf{s}$ are *excluded* from all contexts that are suffixes of $\mathbf{s}$. These exclusions are made explicit in the definition below: $G_{\mathbf{s}\setminus\mathcal{R}}$ denotes the adaptive symbol distribution for context $\mathbf{s}$ that excludes all symbols in set $\mathcal{R}$. (Note that the adaptive distributions $G_{\mathbf{s}}$ and summary multisets $\mathcal{M}_{\mathbf{s}}$ are changing objects whose predictions for symbol x_{N+1} depend on the preceding symbols x_1 ... x_N; as earlier in this chapter, this conditional dependence is left implicit for ease of notation.)

For the generalised escape method with parameters α and β, the probabilistic model induced by PPM can then be stated as follows:

$$G_{\mathbf{s}\setminus\mathcal{R}}(x) \;:=\; \begin{cases} \dfrac{\mathcal{M}_{\mathbf{s}}(x) - \beta}{T_{\mathbf{s}}^{\setminus\mathcal{R}} + \alpha} & \text{if } x \in \mathcal{M}_{\mathbf{s}} \text{ and } x \notin \mathcal{R} \\[2ex] \dfrac{U_{\mathbf{s}}^{\setminus\mathcal{R}} \cdot \beta + \alpha}{T_{\mathbf{s}}^{\setminus\mathcal{R}} + \alpha} \cdot G_{\mathrm{suf}(\mathbf{s})\setminus\mathcal{M}_{\mathbf{s}}}(x) & \text{otherwise,} \end{cases} \tag{6.11}$$

where $T_{\mathbf{s}}^{\setminus\mathcal{R}}$ is the total number of symbol occurrences in $\mathcal{M}_{\mathbf{s}}$ after excluding all symbols in $\mathcal{R}$:

$$T_{\mathbf{s}}^{\setminus\mathcal{R}} \;:=\; \sum_{x \in \mathcal{M}_{\mathbf{s}}} \mathcal{M}_{\mathbf{s}}(x) \cdot \mathbb{1}[x \notin \mathcal{R}] \tag{6.12}$$

and $U_{\mathbf{s}}^{\setminus\mathcal{R}}$ is the number of unique symbols in $\mathcal{M}_{\mathbf{s}}$ that do not occur in $\mathcal{R}$:

$$U_{\mathbf{s}}^{\setminus\mathcal{R}} \;:=\; \sum_{x \in \mathcal{X}} \mathbb{1}[x \in \mathcal{M}_{\mathbf{s}}] \cdot \mathbb{1}[x \notin \mathcal{R}]. \tag{6.13}$$

G_{ε} is the top-level (unigram) distribution, and $G_{\mathrm{suf}(\varepsilon)}$ is defined to be uniform (or some other suitable base distribution over the source alphabet $\mathcal{X}$).

6.4.4 Why the escape mechanism is convenient

The probabilistic model used in the classic variants of PPM is implicitly defined through the algorithm's escape mechanism. The escape mechanism is computationally convenient, because the algorithm can stop searching the trie when it finds a context in which the escape criterion isn't satisfied: whenever possible, the algorithm uses only the longest context to compute the probability of a given symbol, and only "backs off" to the next shorter context when the escape criterion triggers.

Interfacing the escape mechanism to an arithmetic coder can therefore be done with a recursive procedure that encodes (for a given context $\mathbf{s}$) either the symbol itself, or an escape event

followed by recursion on the next shorter context suf(s). Such an implementation might make up to $D + 1$ calls to the arithmetic coder's `storeRegion` function to encode a given symbol.

The downside of this computational shortcut is that not all of the information stored in the trie is taken into account, which can lead to worse symbol predictions and therefore less effective compression. Also, the probability distributions induced by the escape mechanism are somewhat harder to reason about than those defined by cleaner mathematical constructs, such as Pitman–Yor processes.

6.5 Blending

The classic PPM algorithm can be modified to use other forms of probability estimation, different from the original "escape mechanism" and its derivatives. Let's now consider an approach that combines the observations from all context depths, blending their probability distributions (rather than switching between them and applying exclusion rules).

In particular, consider the following construction:

$$G_{\mathbf{s}}(x) \;:=\; \frac{\mathcal{M}_{\mathbf{s}}(x) - \beta}{|\mathcal{M}_{\mathbf{s}}| + \alpha} \cdot \mathbb{1}[x \in \mathcal{M}_{\mathbf{s}}] \;+\; \frac{U_{\mathbf{s}}\beta + \alpha}{|\mathcal{M}_{\mathbf{s}}| + \alpha} \cdot \begin{cases} \dfrac{1}{|\mathcal{X}|} & \text{if } \mathbf{s} = \varepsilon \\ G_{\text{suf}(\mathbf{s})}(x) & \text{otherwise,} \end{cases} \tag{6.14}$$

where $U_{\mathbf{s}}$ denotes the number of unique symbols seen in context $\mathbf{s}$, and $\mathcal{M}_{\mathbf{s}}$ is the context's summary multiset:

$$U_{\mathbf{s}} \;=\; \sum_{x \in \mathcal{X}} \mathbb{1}[x \in \mathcal{M}_{\mathbf{s}}]. \tag{6.15}$$

Equation (6.14) is an instance of construction (4.4) and defines a probabilistic model that corresponds to "interpolated Kneser–Ney smoothing" (Chen and Goodman, 1998) with uniform base distribution, generalised to include a global strength parameter α (originally assumed to be zero). This model can be considered an approximate sequential construction of a Pitman–Yor process; this connection is shown by Teh (2006a).

The probability assignments of equation (6.14), together with the shallow update rule from section 6.3.3, define a PPM-like compressor which I'll refer to as 'BPPM'.

BPPM can be implemented straightforwardly by replacing PPM's escape mechanism with equation (6.14). There are no more `ESC` symbols in BPPM: symbol probabilities are always calculated by visiting contexts of all depths. Computing the cumulative symbol distributions, as required for the arithmetic coder, is computationally more expensive compared to standard PPM, but caching can reduce the costs a bit.

6.5.1 Optimising the parameters of a blending PPM

Having modified PPM's probability assignments to use blending rather than backing off, let's look at some of the properties of the resulting compression algorithm. As BPPM has very similar parameters to those of PPMG, we can investigate its compression effectiveness as a function of D, α, and β and compare it to that of PPMG.

One might hope that, due to the increased sharing of information among different contexts, BPPM has stronger compression effectiveness than PPMG on human text. When BPPM's parameters are set optimally, this is indeed the case: see Tables 6.3 and 6.4 for a direct comparison. However, a blending PPM is not by itself sufficient to guarantee improved compression over a PPM that uses backing off (e.g. with one of the standard escape mechanisms): for some parameter settings, BPPM performs worse than PPMG.

Figure 6.6 shows how BPPM's compression effectiveness changes as a function of the maximal context depth D, for fixed (α, β). Note how the 'dip and rebound' effect is even stronger than the one exhibited by PPMD in Figure 6.2. Finally, Figure 6.7 (page 126) shows a contour plot of BPPM's compression effectiveness as a function of α and β, for fixed D, measured on `alice29.txt` of the Canterbury corpus. Note that the shape of the contours are very different from those in the corresponding plot for PPMG (Figure 6.4, page 120).

File	Size (bytes)	α_{OPT}	β_{OPT}	D_{OPT}	bps
`alice29.txt`	152089	0.4006	0.8605	9	2.0542
`asyoulik.txt`	125179	0.4512	0.8882	10	2.3277
`fields.c`	11150	−0.1933	0.8511	12	1.8315
`lcet10.txt`	426754	0.1831	0.8784	9	1.8198
`plrabn12.txt`	481861	0.6306	0.8183	6	2.2048
`random.txt`	100000	8.7332	0.0000	1	6.0064
`xargs.1`	4227	−0.0372	0.7916	7	2.8021
`world192.txt`	2473400	−0.0703	0.9307	20	1.2211
`shakespeare.txt`	5283795	1.0200	0.9204	10	1.9642

Table 6.4: The best possible compression of BPPM for various corpus files, and the parameters $(\alpha_{\mathsf{OPT}}, \beta_{\mathsf{OPT}}, D_{\mathsf{OPT}})$ for which the optimum is achieved.

6.5.2 Backing-off versus blending

The choice between backing-off (as used in PPM's original escape mechanism) and blending (as used in BPPM) generally has a big impact on the resulting compression effectiveness. Using blending may be computationally more expensive than backing off, as more memory accesses are needed. For non-integer settings of α and β, there may also be an added cost through slightly more costly arithmetic operations.

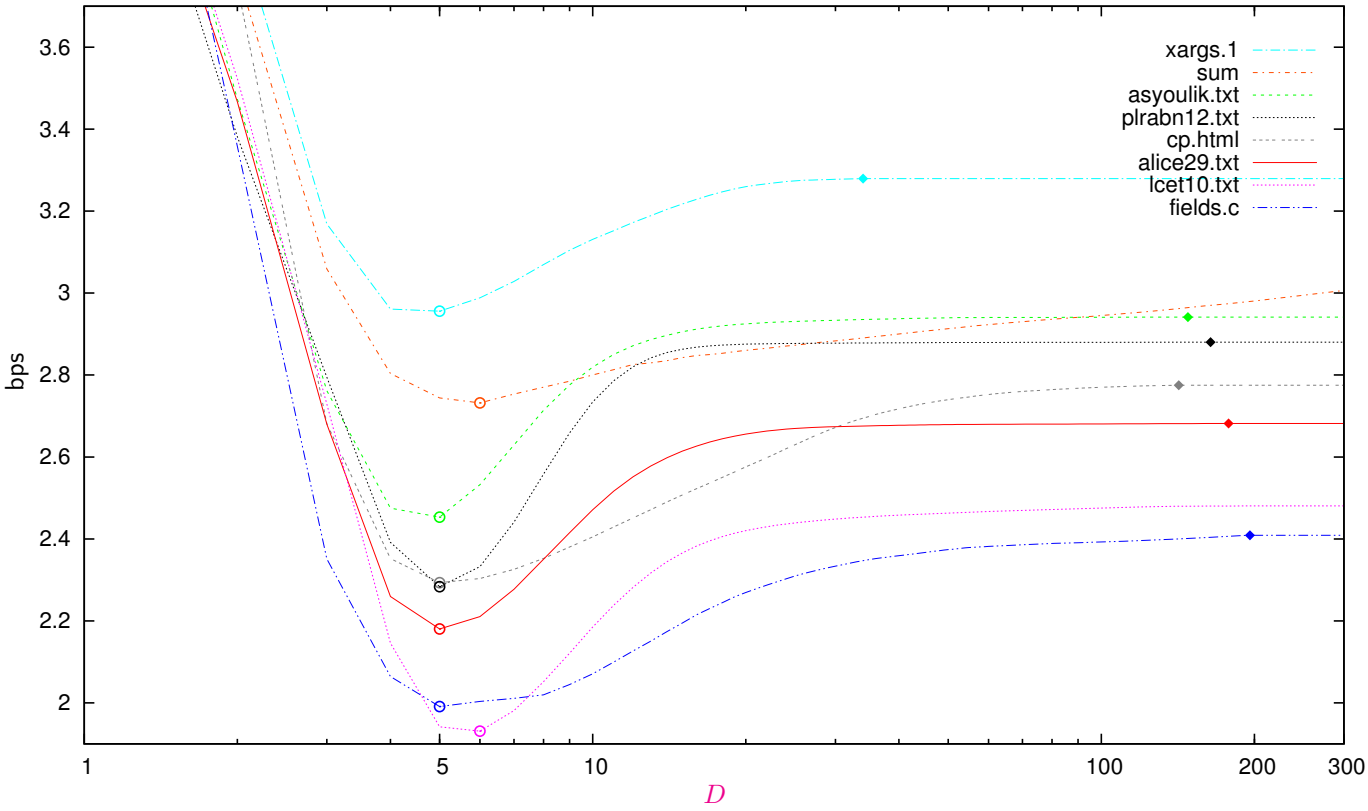


Figure 6.6: BPPM's compression effectiveness on English text in bits per symbol, as a function of D (the maximum context depth). The graph shows that for BPPM with parameters fixed to $\alpha = 0$ and $\beta = \frac{1}{2}$, increasing the context depth doesn't improve the compression effectiveness. For each file, a circle marks the setting for D which yields the most effective compression, and a diamond indicates the depth limit (beyond which added depth has no effect). For `alice29.txt`, BPPM's optimum of 2.1802 bps is reached at $D = 5$, and worsens as depth increases until converging to 2.6819 bps at $D = 178$ (its depth limit). Beyond the depth limit, the compressed file size stays constant (and equals the result produced by UKN-Deplump).

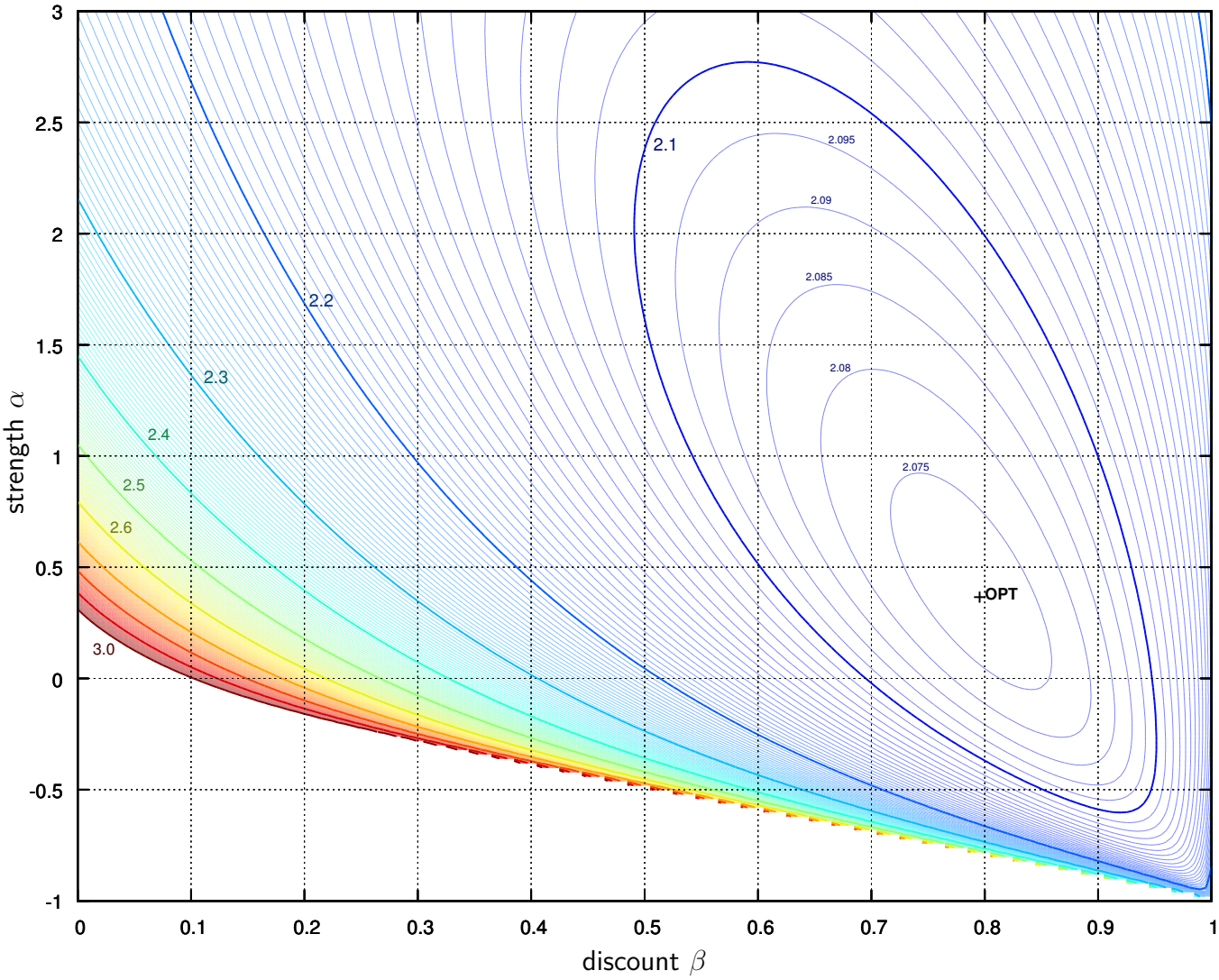


Figure 6.7: Contour plot of BPPM's compression effectiveness on `alice29.txt`, as a function of strength α and discount β, with fixed maximum context depth $D = 6$. The compression effectiveness is measured in bits / symbol.
The optimum of 2.0721 bps is located at $\alpha_{\mathsf{OPT}} \approx 0.366$ and $\beta_{\mathsf{OPT}} \approx 0.796$. Note that BPPM's optimal depth for `alice29.txt` is $D = 8$ (not $D = 6$). The optimal settings of α and β for different choices of D can be found in Table 6.5. Jointly optimised settings of the parameters (α, β, D) for various other corpus files can be found in Table 6.4.

D	α_{OPT}	β_{OPT}	bytes	bps	$\alpha=0,\ \beta=\frac{1}{2}$	$\alpha=\frac{1}{2},\ \beta=\frac{3}{4}$	$\alpha=\frac{1}{2},\ \beta=\frac{17}{20}$
0	—	—	152199	8.0057	8.0057	8.0057	8.0057
1	9.2706	0.0000	86923	4.5721	4.5730	4.5738	4.5742
2	1.8381	0.3286	65893	3.4659	3.4667	3.4695	3.4718
3	0.7982	0.5351	50886	2.6765	2.6803	2.6808	2.6867
4	0.3808	0.6669	42574	2.2393	2.2594	2.2424	2.2525
5	0.3542	0.7480	40157	2.1122	**2.1802**	2.1127	2.1219
6	0.3660	0.7955	39395	2.0721	2.2106	2.0734	2.0773
7	0.3729	0.8265	39173	2.0604	2.2774	**2.0661**	2.0626
8	0.3932	0.8464	39097	2.0564	2.3506	2.0689	2.0570
9	0.4006	0.8605	**39055**	**2.0542**	2.4150	2.0740	**2.0545**
10	0.4091	0.8704	39073	2.0552	2.4711	2.0820	2.0558
11	0.4196	0.8771	39095	2.0563	2.5167	2.0895	2.0577
12	0.4252	0.8822	39110	2.0571	2.5517	2.0954	2.0593
13	0.4306	0.8857	39120	2.0577	2.5779	2.1000	2.0607

Table 6.5: BPPM's compression effectiveness on English text `alice29.txt` (152 089 bytes) for different context depths D and different settings of the global strength and discount parameters. The table shows, for each setting of D, the most effective compression possible with BPPM, and for which setting $(\alpha_{\mathsf{OPT}}, \beta_{\mathsf{OPT}})$ the optimum is obtained. For comparison, the table also shows compression results for parameters fixed at $(\alpha=0,\ \beta=\frac{1}{2})$, $(\alpha=\frac{1}{2},\ \beta=\frac{3}{4})$, and $(\alpha=\frac{1}{2},\ \beta=\frac{17}{20})$.

On most compression corpora, blending produces consistently better results than backing-off. This is also reported by e.g. Bunton (1997) who constructed blending variants of PPMC and PPMD. However, the behaviour of the predictive distribution changes dramatically when backing-off is replaced with blending, significantly altering which settings of α and β result in good compression. This issue has possibly been overlooked in prior work.

Consider for example the settings $(\alpha=0, \beta=\frac{1}{2})$ used by PPMD's escape mechanism: Figure 6.4 shows that (for `alice29.txt`) these values are close to the empirical optimum. They are also convenient values from a computational perspective, allowing an implementation of PPMD to compute the symbol and escape probabilities cheaply, using e.g. bitshifts. But using these same parameters for a blending version of PPM would result in less effective compression of the same text, as suggested by Figure 6.7. A more suitable choice of computationally convenient parameters for blending variants of PPM might be e.g. $\alpha=\frac{1}{2}, \beta=\frac{3}{4}$. This effect is also demonstrated in Table 6.5.

To further illustrate the difference between blending and backing off, Table 6.6 shows a side-by-side comparison of the predictive log probabilities produced by a blending version of PPM versus a backing-off version of PPM. Figures 6.8 and 6.9 show the sequence of predictive log probabilities generated by PPM and BPPM on `alice29.txt`, visualised as a point cloud. The escape mechanism of PPM tends to produce repeated occurrences of particular probability values; these repetitions show up as distinctive horizontal lines in Figures 6.8 and 6.10.

There may be ways of compromising between backing-off and blending. For example, the mechanism used by Shkarin (2001a) in PPMII could be considered such a compromise. The modifications described by Shkarin are extensive and are not discussed further in this chapter.

6.6 Unbounded depth

Each of the PPM variants discussed in the previous sections learns symbol distributions for contexts up to a maximum depth D, where D is often hard-wired into the algorithm. Imposing a maximum context depth can be used to limit the memory requirements of an algorithm; but there may be smarter ways to conserve resources, such as the tree node deletion technique suggested by Bartlett et al. (2010).

As already stated in section 6.4.1, there's a conceptual inelegance in imposing a depth limit. Also, it seems dissatisfying when expanding a model's cognitive horizon ends up worsening (rather than improving) compression, as it does with PPM and BPPM.

This section reviews context-sensitive sequence compressors that use contexts of unbounded depth, and shows how context-sensitive sequence models can be tuned to take advantage of the information from deep contexts. Particular attention is given to the Sequence Memoizer and the Deplump algorithm, and their distinguishing features that make them beat the unbounded-depth compressors based on PPM*.

6.6.1 History

PPM variants with unbounded context lengths have been around for a while. PPM* by Cleary et al. (1995) was the first such method, contributing an algorithmic engine with practical support for contexts of unbounded depth. PPM* used the escape mechanism of PPMC, the "full updates" rule of section 6.3.3, and an added hack that selects the shallowest non-branching node of the current context for computing the probability of the next symbol.

Noting that PPM* failed to beat comparable fixed-depth algorithms, Bunton (1997) improved and generalised the implementation of PPM* in several ways: firstly by building an improved engine with reduced memory footprint. Bunton's engine is very similar to (but independent from) that of Ukkonen (1995). Secondly, Bunton made systematic modifications to the model of PPM* and compared the results. The winning variant (D*XSM) used shallow updates rather than full updates, new state selection rules, blending rather than backing-off, and PPMD's discount and strength parameters (although these are suboptimal for blending, as shown in section 6.5.1). The resulting compression effectiveness surpassed that of PPM* on all standard corpora.

Table 6.6: Symbol-by-symbol comparison of PPM's and BPPM's predictive $\log_2$ probability mass for the example string "`abcdabcdXabcd`". To make the comparison as helpful as possible, both models were initialised to similar parameters: uniform base distribution over an alphabet of 257 symbols (2^8 byte codes + `EOF`), all strengths set to zero, all discounts set to $\frac{1}{2}$, and shallow update rules. Results for LZW are included for curiosity.

Pos	Sym	PPM($\alpha=0, \beta=\frac{1}{2}$)	BPPM($\alpha=0, \beta=\frac{1}{2}$)	LZW
1	a	-8.0056245	-8.0056245	-8.0056245
2	b	-9.0	-9.0056245	-8.0112273
3	c	-8.9943534	-9.0056245	-8.0168083
4	d	-8.9886847	-9.0056245	-8.0223678
5	a	-3.0	-2.9777186	-7.0279060
6	b	-1.0	-0.8604566	-1.0
7	c	-1.0	-0.3670076	-7.0334230
8	d	-1.0	-0.1718648	-1.0
9	X	-9.9829936	-13.3275526	-8.0389190
10	a	-2.0	-1.9906742	-6.4594316
11	b	-0.4150375	-0.3804376	-0.5849625
12	c	-0.4150375	-0.1777148	-1.0
13	d	-0.4150375	-0.0861227	-7.0498485
14	EOF	-9.9772799	-13.4910514	-9.0552824
Total:		-64.1940487	-68.8530991	-80.3058010

Correct implementations of PPM or PPM* (with backing-off and symbol exclusions, escape method D, shallow updates, no state selection rules or other modifications) should exactly reproduce the numbers on the left. Blending variants such as BPPM or UKN-Deplump / Sequence Memoizer (with correct parameters, shallow updates, no online optimisations or other modifications) should exactly reproduce the numbers of the middle column. For finite-depth methods, any depth $D \geq 2$ should produce these values.
Some characteristics of the algorithms are visible in the table:

- Both algorithms compress the first symbol "`a`" using the uniform base distribution: $\log_2 \frac{1}{257} \approx -8.0056245$.
- For the first occurrence of characters `bcd`, the difference between "blending" versus "backing off with symbol exclusions" is visible. Both algorithms pay a penalty for unseen symbols: in a context with only one observed symbol, that penalty equals exactly 1 bit (for $\alpha = 0$ and $\beta = \frac{1}{2}$). The penalty is added to the new symbol's probability mass from the base distribution; BPPM uses the base distribution unmodified, giving mass -9.0056 to each symbol. PPM too uses the base distribution, but excludes the mass of symbols that were seen before, giving diminishing masses -9.0, -8.9943 and -8.9886.

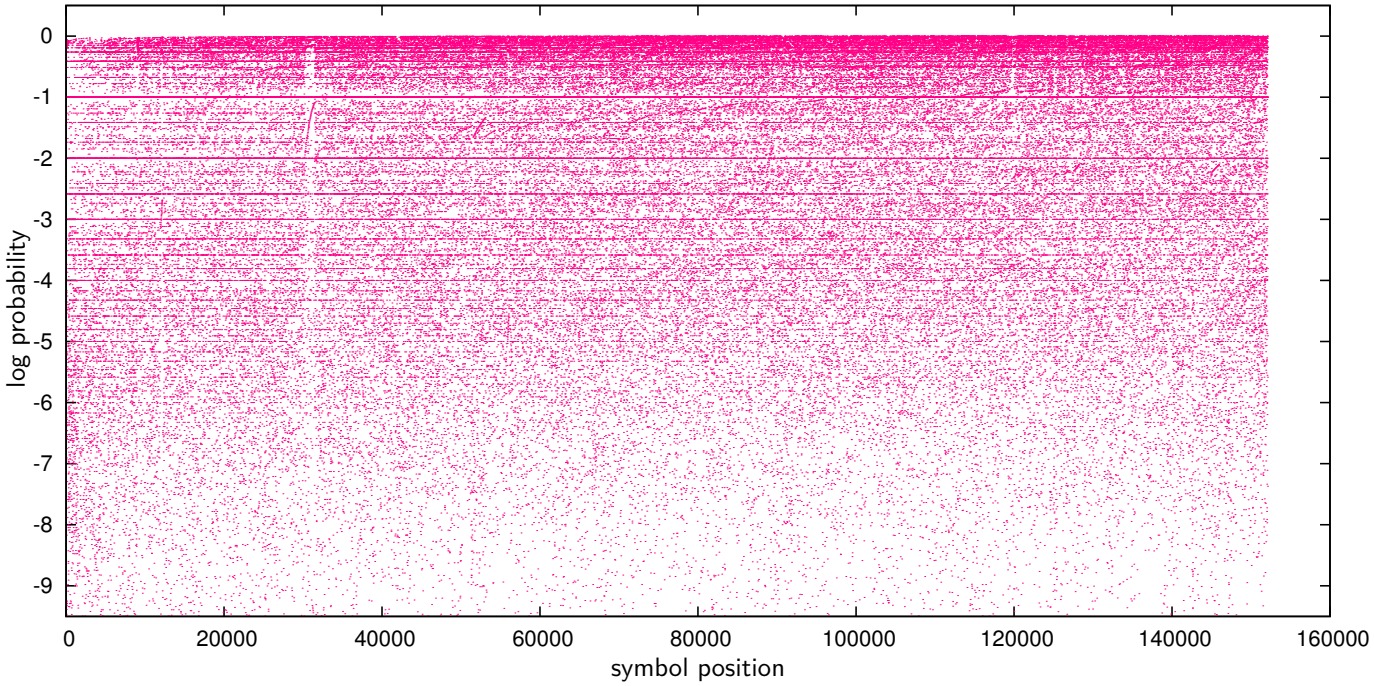


Figure 6.8: Symbol-wise log probabilities produced by PPMG (backing off, $\alpha = 0$, $\beta = \frac{1}{2}$, $D = 5$) for every symbol in `alice29.txt`. The horizontal lines are characteristic of PPM's backing off escape mechanism.

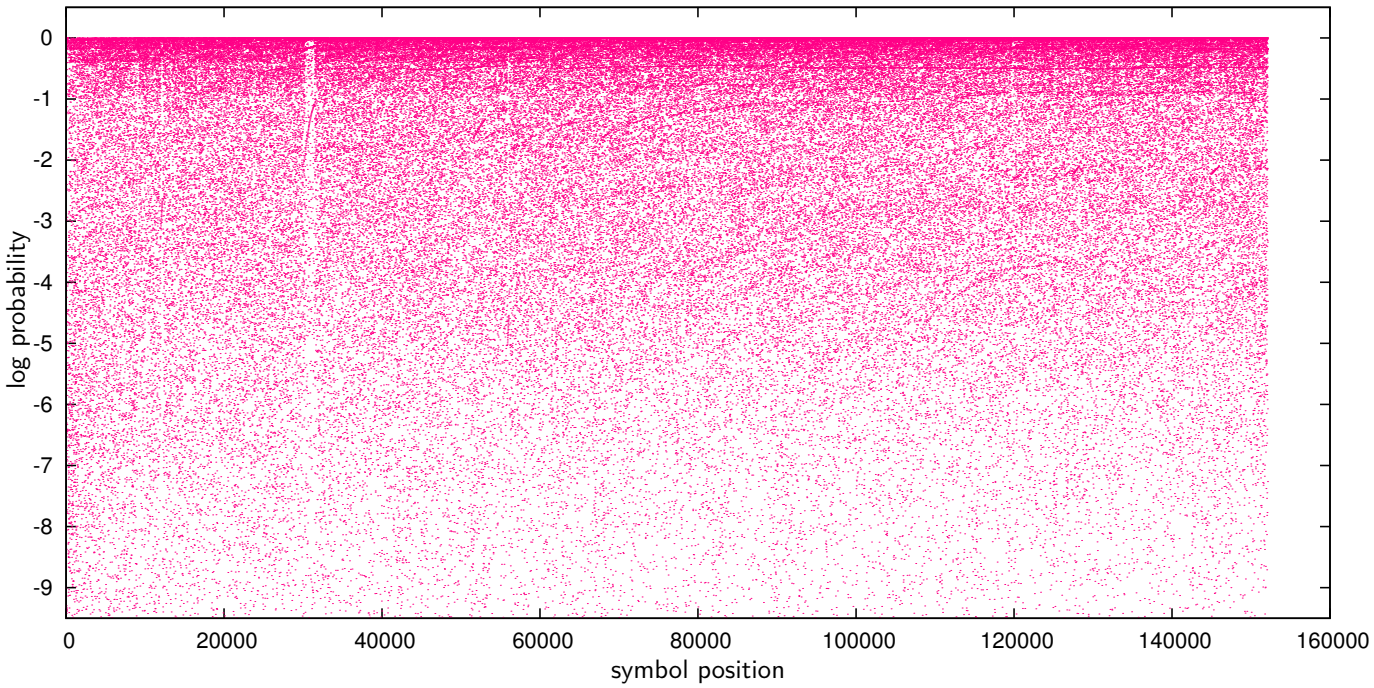


Figure 6.9: Log probabilities produced by BPPM (blending, $\alpha = 0$, $\beta = \frac{1}{2}$, $D = 5$) for every symbol in `alice29.txt`. The blending mechanism produces a much smoother histogram, with slightly heavier tails (not visible in this plot).

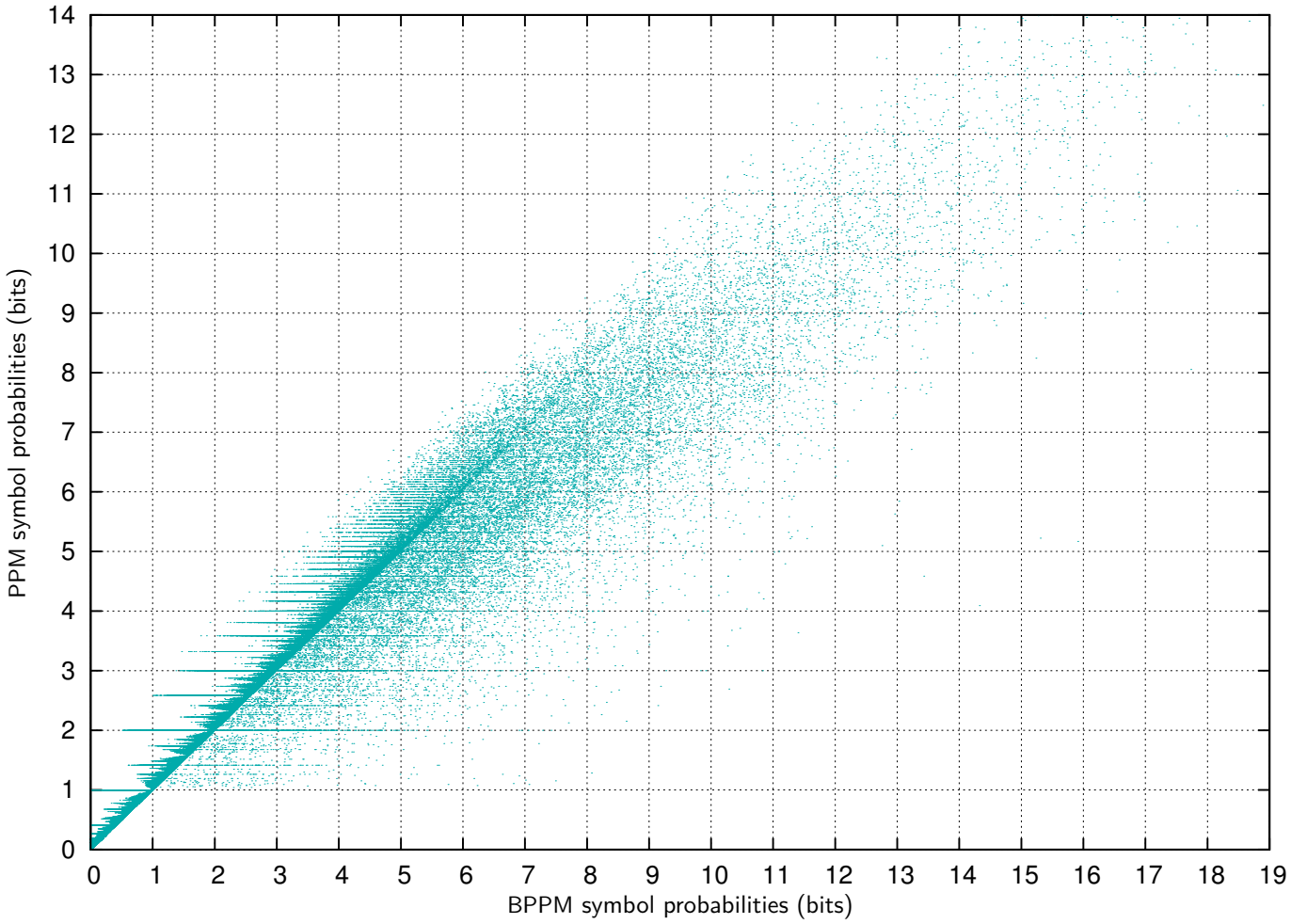


Figure 6.10: Scatter plot comparing the symbol-wise predictions of PPM (from Figure 6.8) with those of BPPM (from Figure 6.9) for `alice29.txt`. Each dot in the plot represents one of the 152 089 symbols in `alice29.txt`. Both algorithms used settings ($\alpha = 0$, $\beta = \frac{1}{2}$, $D = 5$), and differed only in their blending / backing off.

Gasthaus et al. (2010) and Bartlett and Wood (2011) designed Deplump, a new unbounded-depth context compressor based on the probabilistic model of the Sequence Memoizer by Wood et al. (2009, 2011). Deplump's data structure takes inspiration from the work of Ukkonen (1995), but operates differently from existing PPM engines in that it stores the input string in reverse, and finds each symbol context by traversing the tree from the root. With carefully chosen parameters, Deplump beats the compression effectiveness of all unbounded-depth compressors that preceded it.

6.6.2 The Sequence Memoizer

The variants of PPM presented so far were designed mainly from an algorithmic point of view, and their implicit probabilistic models were rarely stated explicitly, and perhaps more of an afterthought.

A radically different approach is taken in the work of Wood et al. (2009), where the probabilistic model is constructed first, and corresponding sequential prediction and learning algorithms are derived afterwards. The model, named *the Sequence Memoizer*, applies an unbounded-depth hierarchical Pitman–Yor process prior to discrete sequences, and gives an incremental, sequential construction of the conditional symbol distributions using approximate Bayesian inference algorithms.

Like PPM and PPM*, the Sequence Memoizer can use a search tree to store its sufficient statistics conveniently. An elegant feature of the Sequence Memoizer model is that this tree can be represented in a space-saving manner without altering the model: by placing careful restrictions on the parameters of the hierarchical Pitman–Yor process, the Sequence Memoizer allows the sufficient statistics of non-branching nodes to be marginalised out exactly, making it unnecessary to store them. This marginalisation exploits fragmentation and coagulation properties of the hierarchical Pitman–Yor process (Pitman, 1999; Ho et al., 2006). For details, the reader is encouraged to read the papers by Wood et al. (2009, 2011), Gasthaus and Teh (2010), and Bartlett et al. (2010).

6.6.3 Hierarchical Pitman–Yor processes

The Pitman–Yor process is based on work by Perman et al. (1992); Pitman and Yor (1995), and can be considered a two-parameter generalisation of the Dirichlet process. Teh (2006b) describes a sequential construction for *hierarchical* Pitman–Yor processes and matching approximate learning mechanisms. When used for context-sensitive data compression, the pre-

dictive distributions of a hierarchical Pitman–Yor process can be stated as follows:

$$G_{\mathbf{s}}(x) := \frac{\mathcal{M}_{\mathbf{s}}(x) - \mathcal{T}_{\mathbf{s}}(x) \cdot \beta}{|\mathcal{M}_{\mathbf{s}}| + \alpha} + \frac{|\mathcal{T}_{\mathbf{s}}| \cdot \beta + \alpha}{|\mathcal{M}_{\mathbf{s}}| + \alpha} \cdot \begin{cases} \frac{1}{|\mathcal{X}|} & \text{if } \mathbf{s} = \varepsilon \\ G_{\text{suf}(\mathbf{s})}(x) & \text{otherwise} \end{cases} \tag{6.16}$$

where $\mathcal{T}_{\mathbf{s}}$ is a submultiset of $\mathcal{M}_{\mathbf{s}}$ that records, for each value $x \in \mathcal{X}$, how often x was generated from the second mixture component (involving the parent distribution $G_{\text{suf}(\mathbf{s})}$), out of the $\mathcal{M}_{\mathbf{s}}(x)$ times x was generated by $G_{\mathbf{s}}$ in total. Equation (6.16) is an instance of construction (4.5).

The "correct" Bayesian learning mechanism for a sequentially constructed hierarchical Pitman–Yor process does not have an exact analytic form, as the $\mathcal{T}_{\mathbf{s}}$ cannot be determined exactly from observed data. (Neither can the $\mathcal{M}_{\text{suf}(\mathbf{s})}$ or $\mathcal{T}_{\text{suf}(\mathbf{s})}$.) Approximate inference algorithms exist, and are described by Teh (2006b). For the Sequence Memoizer in particular, Gasthaus et al. (2010) describe and compare suitable approximate inference methods.

If $\mathcal{T}_{\mathbf{s}}$ is restricted to be a set (rather than a multiset), equation (6.16) turns into the predictive distribution of BPPM shown in equation (6.14).

6.6.4 Deplump

The Sequence Memoizer was first applied to data compression by Gasthaus et al. (2010), and subsequently refined and made practical by Bartlett and Wood (2011); the resulting algorithm was branded "Deplump". The emphasis of this algorithm was to stay as true as possible to the original Sequence Memoizer model, while making every effort to make compression computationally efficient.

The computational innovations include an algorithmic engine that supports unbounded context depths and collapses non-branching nodes in a way that takes advantage of the Sequence Memoizer's marginalisation properties. Gasthaus et al. (2010) describe two different approximate inference algorithms that can be used for updating the data structure: one of them (1PF) makes stochastic updates to the $\mathcal{T}_{\mathbf{s}}$ counts using a particle filter, the other (UKN) uses equation (6.14) and the shallow update rule from section 6.3.3. This latter approach has the benefit that the $\mathcal{T}_{\mathbf{s}}$ become a deterministic function of $\mathcal{M}_{\text{suf}(\mathbf{s})}$ and therefore don't need to be stored explicitly in the tree. This UKN variant of Deplump is functionally identical to BPPM with unbounded context depth, with the exception that BPPM places fewer restrictions on its parameters.

Improvements by Gasthaus and Teh (2010) and Bartlett et al. (2010) reduced the memory footprint of the Sequence Memoizer machinery, for example by carefully deleting nodes at random from the tree when a given memory limit is exceeded. These changes were integrated

into Deplump by Bartlett and Wood (2011), and augmented with tweaks that departed from the original model, such as online optimisation of the discount parameters during compression.

The compression effectiveness on standard corpora reported for the Deplump algorithms show a promising improvement over those of most existing PPM variants, one notable exception being PPMII by Shkarin (2001a).

The compressors based on the Sequence Memoizer bear a remarkable similarity to other PPM compressors, and should be considered members of the same family. For example, PPMG can be turned into an instance of UKN-Deplump by replacing backing-off with blending, and adding support for unbounded context depth.

6.6.5 What makes Deplump compress better than PPM?

One of the claims by Gasthaus et al. (2010) is that Deplump's superior compression effectiveness (over PPM, PPM* and CTW) stems from Deplump's ability to make effective use of data from contexts of unbounded depth, and from using a probabilistic model that makes power-law assumptions. But most PPM compressors make power-law assumptions, too, including unbounded-depth variants such as PPM* by Cleary et al. (1995) or Bunton (1997). So one might wonder where Deplump's ability to harness contexts of unbounded depth *really* comes from. After all, Deplump is not all that different from BPPM, and section 6.5.1 and Figure 6.6 showed how increased context depth worsens BPPM's compression effectiveness rather than improving it.

So how does Deplump differ? The secret lies in the way the parameters are chosen. BPPM (as it was described in section 6.5) uses one pair of parameters (α, β) that is shared by all contexts. Deplump instead uses *depth-dependent* parameters, i.e. a pair of parameters (α_d, β_d) for each depth d, shared by all contexts of that depth.[4] If these depth-dependent parameters are set to carefully chosen values, the "dip-and-rebound" effect (visible in e.g. Figure 6.6 on page 125) can be avoided. An example is shown in Figure 6.11: for a BPPM with carefully chosen depth-dependent parameters, additional context depth does not significantly worsen compression.

However, even with carefully set parameters, unbounded depth does not necessarily improve compression. For example, Table 6.7 compares the compression effectiveness of unbounded-depth Deplump to that of variants with imposed depth limits; in each case, Deplump's depth-dependent discount parameters were set to the default values from Gasthaus (2010). As can be seen from the table, truncating Deplump's depth can improve compression.

[4] Actually, to preserve the marginalisation properties of the Sequence Memoizer, only the discount parameters are depth-dependent in Deplump. This is less of a concern for BPPM, where strength parameters can be made depth-dependent, too.

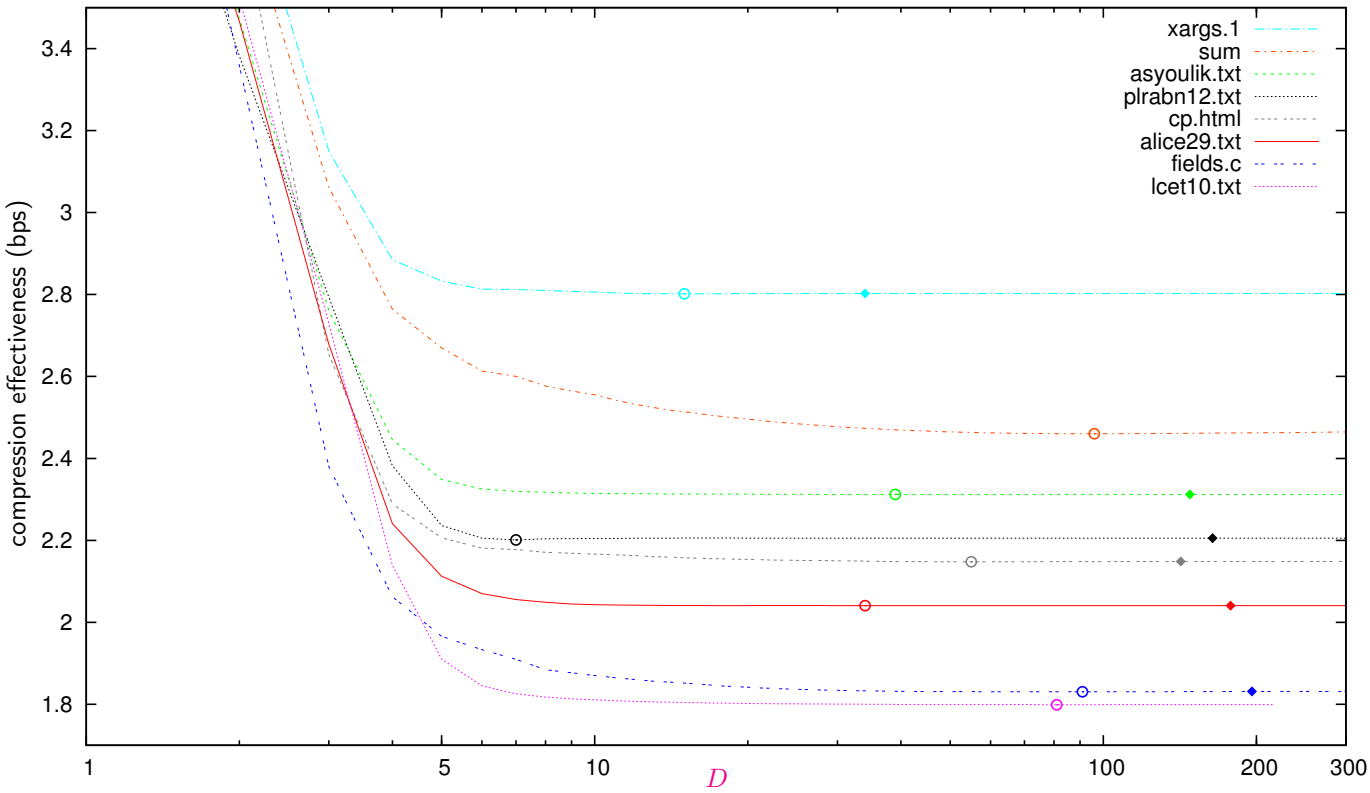


Figure 6.11: Compression effectiveness of BPPM with depth-dependent parameters, as a function of D (the maximal context depth). Results are shown for various files of the Canterbury corpus, in bits per symbol. A circle indicates the optimal maximal context depth, and a solid diamond marks the context depth horizon, beyond which the slope of the curve is zero. The following depth-dependent strength and discount parameters were used:

$$(\alpha_1 \ldots \alpha_{10}) = (20.7135, 0.8211, 0.4377, -0.1019, 0.1752, 0.0654, 0.6838, 0.5443, 0.6390, 0.8851)$$
$$(\beta_1 \ldots \beta_{10}) = (0.0002, 0.5560, 0.7375, 0.7916, 0.8707, 0.8905, 0.9264, 0.9386, 0.9464, 0.9216)$$

The deepest parameters $(\alpha_{10}, \beta_{10})$ were used for all $D \geq 10$.
Compare this plot to e.g. Figure 6.6 (on page 125) or Figure 6.2 (on page 118).

Table 6.7: Comparison between unbounded-depth and fixed-depth UKN-Deplump's compression effectiveness (in bits per symbols). Deplump's discount parameters were set to the default values found in the source code of Gasthaus (2010): $\beta_1 \ldots \beta_5 = (0.62, 0.69, 0.74, 0.80, 0.95)$ where $\beta_D = \beta_5$ for all $D > 5$. Deplump's global strength parameter was set to $\alpha = 0$.

File	$D=8$	$D=10$	$D=15$	$D=20$	$D=30$	$D=40$	$D=50$	$D=75$	$D=100$	$D=\infty$
alice29.txt	2.056	2.050	**2.048**	2.048	2.048	2.048	2.048	2.048	2.048	2.049
asyoulik.txt	2.320	2.316	2.315	2.314	**2.314**	2.314	2.314	2.314	2.314	2.314
cp.html	2.183	2.173	2.162	2.157	2.154	2.153	**2.153**	2.154	2.154	2.154
fields.c	1.889	1.867	1.846	1.836	1.829	1.827	**1.827**	1.827	1.827	1.829
grammar.lsp	2.273	2.249	2.238	2.234	2.232	2.232	**2.232**	2.234	2.234	2.234
kennedy.xls	**1.570**	1.582	1.615	1.615	1.615	1.615	1.615	1.615	1.615	1.615
lcet10.txt	1.825	1.817	1.810	1.808	1.807	1.807	1.807	**1.806**	1.806	1.806
plrabn12.txt	**2.212**	2.214	2.216	2.216	2.215	2.215	2.215	2.215	2.215	2.215
ptt5	0.786	**0.781**	0.784	0.784	0.786	0.790	0.795	0.794	0.795	fail
sum	2.564	2.534	2.488	2.470	2.453	2.446	2.443	2.441	**2.440**	2.450
xargs.1	2.824	2.818	**2.812**	2.812	2.814	2.812	2.812	2.812	2.812	2.812

The idea of using depth-dependent discount parameters may have originated with Chen and Goodman's versions of Kneser–Ney smoothing, in particular the "interpolated Kneser–Ney" method from section 4.1.6 of their technical report (Chen and Goodman, 1998).

6.6.6 Optimising depth-dependent parameters

Selecting discount parameters based on context depth is the primary contribution to the compression effectiveness of Deplump and the Sequence Memoizer. But the big question is how these depth-dependent parameters should be set. This problem is somewhat similar to optimising the escape mechanism for classic PPM, but harder because there are more parameters.

One way of optimising the parameters is through the use of a conjugate gradient minimization algorithm: using such a method requires being able to evaluate the *gradients* of the model's predictive log probability, i.e. its partial derivatives with respect to each of the parameters. One of the benefits of working explicitly with probability distributions is that the gradients are often easy to obtain analytically. All optimisations reported in this chapter were carried out using a custom built conjugate gradient search algorithm based on `macopt` (MacKay, 2002), using analytically derived gradients for the depth-dependent discount and strength parameters.

One might also wonder how many depth-dependent parameters are required to get a decent improvement in compression effectiveness. Table 6.8 shows the best possible compression of file `alice29.txt` as a function of D and W, where D is the maximum context depth, and W is the number of depth levels d that should receive their own parameter set (α_d, β_d). The results suggest that data from deep contexts are used most effectively when at least $W = 7$ depth-dependent sets of optimised parameters are used (for depths $0 \leq d < W$). The table also shows that even though it pays off to increase the context depth, there are diminishing returns.

A technique that is mentioned (but not described in detail) in the papers by Gasthaus et al. (2010) and Bartlett and Wood (2011) is to gradually optimise the parameters "online", i.e. after each symbol, interleaved with updating counts in the data structure. Adding an online optimisation method to a probabilistic model is a fundamental modification of that model, and should probably be noted as such. Online optimisations could also be applied to other compression methods, but are not investigated further in this chapter.

Table 6.8: Best possible compression effectiveness of BPPM, measured on `alice29.txt`, as a function of context depth D and the number W of depth-dependent strength and discount parameters. For each setting of D and W, BPPM's depth-dependent discount and strength parameters (W each) were jointly optimised using a conjugate-gradient method. The table shows the compression effectiveness, in bits per symbol, at the optimum parameter setting. The column for $W=1$ contains the results from Table 6.5 (on page 127).

D	$W=1$	$W=2$	$W=3$	$W=4$	$W=5$	$W=6$	$W=7$	$W=8$	$W=9$	$W=10$
1	4.5721									
2	3.4659	3.4657								
3	2.6765	2.6761	2.6761							
4	2.2393	2.2387	2.2386	2.2383						
5	2.1122	2.1113	2.1107	2.1107	2.1105					
6	2.0721	2.0711	2.0700	2.0694	2.0686	2.0685				
7	2.0604	2.0593	2.0577	2.0565	2.0547	2.0546	2.0545			
8	2.0564	2.0552	2.0533	2.0513	2.0485	2.0482	2.0481	2.0479		
9	**2.0542**	**2.0530**	**2.0507**	**2.0482**	2.0444	2.0438	2.0437	2.0436	2.0435	
10	2.0552	2.0538	2.0514	2.0484	2.0438	2.0429	2.0427	2.0427	2.0426	2.0425
11	2.0563	2.0550	2.0524	2.0489	2.0437	2.0425	2.0422	2.0422	2.0422	2.0421
12	2.0571	2.0557	2.0530	2.0492	2.0434	2.0420	2.0417	2.0417	2.0416	2.0416
13	2.0577	2.0563	2.0534	2.0495	**2.0433**	2.0417	2.0413	2.0413	2.0413	2.0412
14	2.0583	2.0569	2.0540	2.0499	2.0434	2.0416	2.0411	2.0411	2.0411	2.0411
15	2.0588	2.0573	2.0544	2.0501	2.0433	2.0414	2.0409	2.0409	2.0409	2.0408
16	2.0592	2.0577	2.0547	2.0503	2.0433	2.0413	2.0408	2.0407	2.0407	2.0407
17	2.0595	2.0581	2.0550	2.0505	2.0434	2.0413	2.0407	2.0406	2.0406	2.0406
18	2.0598	2.0583	2.0553	2.0507	2.0434	2.0412	2.0406	2.0405	2.0405	2.0405
19	2.0601	2.0586	2.0555	2.0509	2.0435	2.0412	2.0405	2.0405	2.0405	2.0405
20	2.0603	2.0588	2.0557	2.0510	2.0435	2.0412	2.0405	2.0405	2.0404	2.0404
21	2.0605	2.0590	2.0559	2.0512	2.0436	2.0412	2.0405	2.0405	2.0404	2.0404
:	:	:	:	:	:	:	:	:	:	:
26	2.0611	2.0596	2.0564	2.0516	2.0437	2.0413	2.0405	2.0404	2.0403	2.0403
31	2.0611	2.0596	2.0564	2.0515	2.0436	2.0410	2.0402	2.0401	2.0401	2.0400
41	2.0612	2.0598	2.0565	2.0516	2.0436	2.0410	2.0401	2.0401	2.0400	2.0400
51	2.0613	2.0598	2.0566	2.0516	2.0436	2.0410	2.0401	2.0400	2.0400	2.0399
61	2.0614	2.0599	2.0566	2.0517	2.0436	2.0410	2.0401	2.0400	2.0399	2.0399
71	2.0614	2.0599	2.0566	2.0517	2.0436	2.0410	2.0401	2.0400	2.0399	2.0399
81	2.0614	2.0599	2.0567	2.0517	2.0436	**2.0410**	2.0401	2.0399	2.0399	2.0398
91	2.0614	2.0599	2.0567	2.0517	2.0436	2.0410	**2.0401**	**2.0399**	2.0399	2.0398
101	2.0614	2.0599	2.0567	2.0517	2.0436	2.0410	2.0401	2.0399	**2.0399**	2.0398
121	2.0615	2.0600	2.0567	2.0517	2.0436	2.0410	2.0401	2.0400	2.0399	**2.0398**

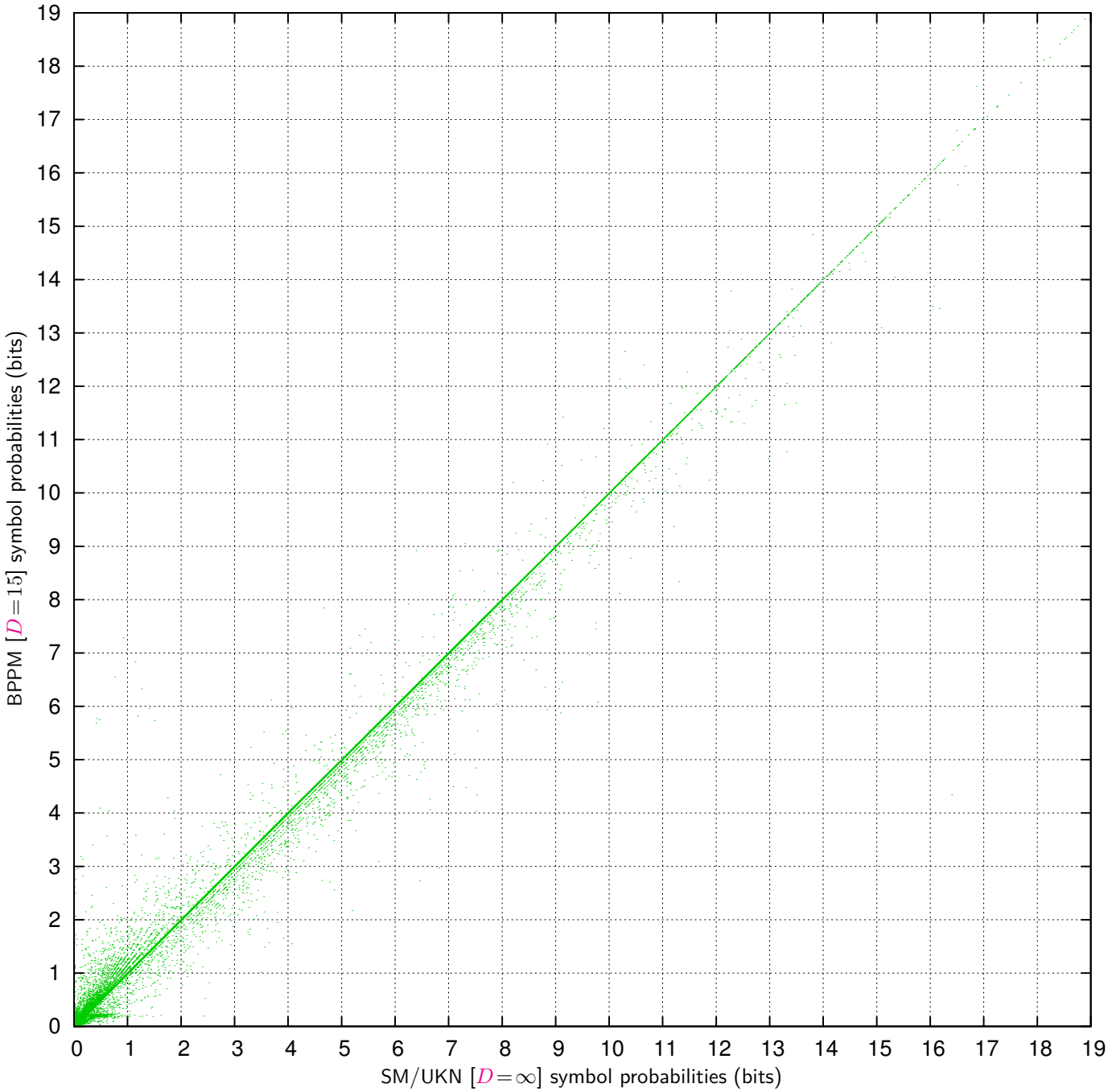


Figure 6.12: Scatter plot comparing the symbol-wise predictions of unbounded-depth UKN-Deplump with those of a finite-depth BPPM (with $D = 15$), on file `alice29.txt` of the Canterbury corpus. Each dot in the plot represents one of the 152 089 symbols in the file. Both algorithms used the same strength and depth-dependent discount parameters:

$$\alpha = 0,\ \beta_1 \ldots \beta_6 = (0.62, 0.69, 0.74, 0.80, 0.95).$$

These parameter settings are the defaults used in the source code of Gasthaus (2010). The plot shows that the predictions of UKN-Deplump are extremely similar to those of a finite-depth BPPM. The predictions converge as D increases, and are identical when $D \geq 178$. BPPM at $D = 15$ compresses `alice29.txt` better than UKN-Deplump by 67 bits (9 bytes).

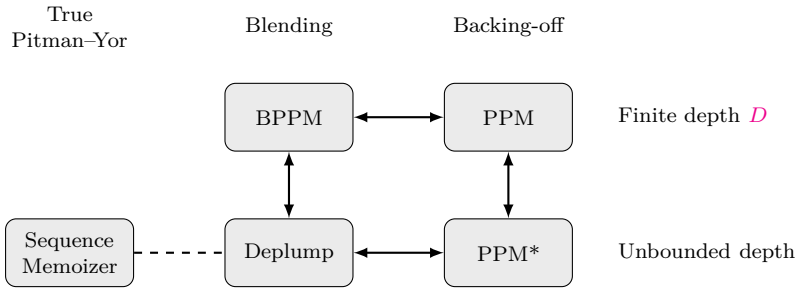


Figure 6.13: The relations between different context-sensitive compression models, ignoring added hacks. These models have similar parameters (strength α and discount β), which can be set globally or made to depend on context depth.

6.6.7 Applications of PPM-like algorithms

From its original conception, the PPM algorithm was designed for data compression of English language text files. But more generally, PPM provides an adaptive predictive distribution over symbols in a sequence, which can be used in other ways.

For example, PPM has been used for text categorisation, e.g. by Frank et al. (2000) and Teahan and Harper (2001): Categories are represented by a collection of documents, with an associated PPM model pre-trained on those documents. New documents are assigned to one of the categories based on which pre-trained PPM predicts it best (i.e. compresses it to the smallest number of bits). PPM can also be used for text correction (Teahan et al., 1998). An algorithm similar to BPPM has found use in the predictive text entry method Dasher (Ward et al., 2000).

Finally, PPM-like prediction models find use in context-sensitive *ensemble compressors* such as PAQ (Mahoney, 2002, 2005). Ensemble compressors combine the predictions from several probabilistic models to form a single predictive distribution that is used for compressing the next symbol. Each of the component models may be context-sensitive, for a more liberal notion of context (any function of the preceding sequence). The mechanism that combines the different models is itself adaptive and context-sensitive, and well worth studying; see e.g. Knoll and de Freitas (2012) for an insightful analysis. Ensemble compressors currently produce the best known compression results on standard corpora, but at a high runtime cost that scales with the number of component models used: typical implementations are orders of magnitude slower than algorithms from the PPM family.

6.7 Conclusions

This chapter reviewed a family of algorithms that include the PPM algorithm by Cleary and Witten (1984a) and compressors based on the Sequence Memoizer by Wood et al. (2009, 2011).

The algorithms in this family differ functionally only in their predictive distributions and their learning mechanism. The algorithmic engines of the compressors are nearly functionally equivalent, but may exhibit different memory and runtime costs.

The predictive distributions of two types of smoothing methods were considered: *backing off* as defined through PPM's escape mechanism, and *blending*, using the predictive distribution corresponding to interpolated Kneser–Ney smoothing. Both of these methods can be parametrised by a global strength parameter α and a discount parameter β. Optimal settings of these parameters were reported on various input files from the Canterbury corpus, both for blending and backing-off.

In practice, good global parameter settings for *backing-off* are $\alpha = 0$ and $\beta = \frac{1}{2}$, corresponding to escape method D by Howard (1993). For *blending*, I recommend setting $\alpha = \frac{1}{2}$ and $\beta = \frac{17}{20}$ (or $\beta = \frac{3}{4}$, if avoiding multiplication and division operations is important).

This chapter also investigated the influence of context depth on compression effectiveness, in particular whether unbounded contexts give a consistent advantage over bounded contexts. For the models that use one global pair of parameters (α, β), unbounded contexts worsen rather than improve the compression effectiveness on human text. One way these models can take advantage of data from deep contexts is by using separate discount (and maybe also strength) parameters that depend on the context depth. These depth-dependent parameters must be set carefully. Models that support contexts of unbounded depth seem more elegant than finite-depth models, but the unboundedness comes at a computational cost. On human text, unbounded-depth contexts can help, but offer only limited returns.

A notable algorithm investigated in this chapter is Deplump by Gasthaus et al. (2010) and Bartlett and Wood (2011). Deplump is essentially a blending PPM variant with support for contexts of unbounded depth; its compression effectiveness on files of the Calgary and Canterbury corpora stems from carefully optimised depth-dependent discount parameters rather than its ability to use contexts of unbounded depth. Even with carefully optimised parameters, there are diminishing returns for unbounded depth; similar conclusions are drawn by Wood (2011). Truncating Deplump's search tree at a fixed depth (e.g. $D = 15$) produces very similar compression results on the files of the Canterbury corpus as the unbounded-depth version.

Future directions. Imposing resource constraints on a compression algorithm is necessary for the method to be of practical use; for PPM-like algorithms, capping memory usage and trie search time are of primary concern. Rather than restricting the maximum context depth, other ways of limiting resource usage might offer better compromises, such as the technique proposed by Bartlett et al. (2010). Although blending is computationally more expensive than backing off, its overhead might be reducible through stochastic caching: such a technique is used for example in PPMII by Shkarin (2001a).

Blending variants of PPM do offer a significant improvement over variants that use backing off. For both fixed and unbounded-depth models, compression effectiveness improves even more when sensibly chosen depth-dependent parameters are used. Further improvement might be gained by making parameters also depend on 'node fanout', i.e. the number of unique symbols seen in a context: such a method is proposed by Chen and Goodman (1996) as "modified interpolated Kneser–Ney" smoothing, and is also used in Shkarin's PPMII. An innovative technique introduced by Deplump is the use of online parameter optimisation: such techniques could find use in many other compression algorithms.

Chapter 7

Multisets of sequences

This chapter shows a slightly more complex application of structural compression: compressing a multiset of strings (e.g. a multiset of words or hash sums) in a way that takes advantage of the multiset's disordered structure. The solution presented in this chapter involves mapping the collection to an order-invariant representation (e.g. a tree), deriving an adaptive probabilistic model for this representation, and then compressing the representation using the model. The resulting method can store a collection of sequences more compactly than the concatenation of the sequences, even when the sequences in the collection are individually incompressible (such as cryptographic hash sums). The effectiveness of the algorithm is demonstrated practically on multisets of SHA-1 hashes, and on multisets of arbitrary, individually encodable objects.

7.1 Introduction

Consider a collection $\mathcal{W}$ of N words $\{w_1 \ldots w_N\}$, each composed of a finite sequence of symbols. The members of $\mathcal{W}$ have no particular ordering (the labels w_n are used here just to describe the method). Such collections occur in e.g. *bag-of-words* models. The goal is to compress this collection in such a way that no information is wasted on the ordering of the words.

Making an order-invariant representation of $\mathcal{W}$ could be as easy as arranging the words in some sorted order: if both the sender and receiver use the same ordering, zero probability could be given to all words whose appearance violates the agreed order, reallocating the excluded probability mass to words that remain compatible with the ordering. However, the correct probability for the next element in a sorted sequence is expensive to compute, making this approach unappealing.

It may seem surprising at first that $\mathcal{W}$ can be compressed in a way that does *not* involve encoding or decoding its components w_n in any particular order. The solution presented in this chapter is to store them "all at once" by transforming the collection to an order-invariant tree representation, deriving an adaptive probabilistic model for this representation, and then compressing the tree using the model.

The problem of compressing collections of sequences has received prior attention in the literature. Reznik (2011) gives a concrete algorithm for compressing *sets* of sequences, also with a tree as latent representation, using an enumerative code (Zaks, 1980; Cover, 1973) for compressing the tree shape. Noting that Reznik's construction isn't fully order-invariant, Gripon et al. (2012) propose a slightly more general tree-based coding scheme for multisets.

The work in this chapter offers a different approach: it derives the exact distribution over multisets from the distribution over source sequences, and factorises it into conditional univariate distributions that can be encoded with an arithmetic coder. It also gives an adaptive, universal code for the case where the exact distribution over sequences is unknown.

7.2 Collections of fixed-length binary sequences

Suppose we want to store a multiset of fixed length binary strings, for example a collection of hash sums. The SHA-1 algorithm (NIST, 1995) is a file hashing method which, given any input file, produces a rapidly computable, cryptographic hash sum whose length is exactly 160 bits. Cryptographic hashing algorithms are designed to make it computationally infeasible to change an input file without also changing its hash sum. Individual hash sums can be used, for example, to detect if a previously hashed file has been modified (with negligible probability of error), and collections of hash sums can be used to detect if a given file is one of a preselected collection of input files.[1]

Each bit digit in a SHA-1 hash sum is uniformly distributed, which renders single SHA-1 sums incompressible. It may therefore seem intuitive at first that storing N hash sums would cost exactly N times as much as storing one hash sum. However, an *unordered* collection of SHA-1 sums can in fact be stored more compactly. The potential saving for a collection of N random hash sums is roughly $\log_2 N!$ bits. For example, the practical savings for a collection of 5000 SHA-1 sums amount to 10 bits per SHA-1 sum, i.e. each SHA-1 sum in the collection takes only 150 bits of space (rather than 160 bits).

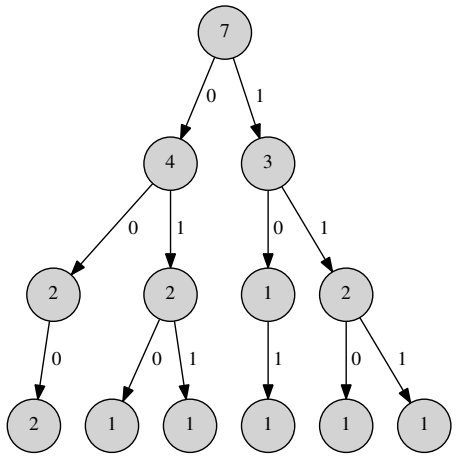
A concrete method for compressing multisets of fixed-length bit strings (such as collections of SHA-1 sums) is described below. The method exploits three properties: firstly, that the

[1] If an application cares mainly about testing membership in a collection, even more compact methods exist, for example Bloom filters (Bloom, 1970). Bloom filters are appropriate when a not-so-negligible chance of false positives is acceptable.

length of each bit string is known and identical for all strings; secondly, that the bits in each string are uniformly distributed; and thirdly, that the order of the strings relative to each other in the collection is not important.

7.2.1 Tree representation for multisets of fixed-length strings

A multiset of binary sequences can be represented with a binary tree whose nodes store positive integers. Each node in the binary tree partitions the multiset of sequences into two submultisets: those sequences whose next symbol is a `0`, and those whose next symbol is a `1`. The integer count n stored in the root node represents the total size of the multiset, and the counts n_0, n_1 stored in the child nodes indicate the sizes of their submultisets. An example of such a tree and its corresponding multiset is shown in Figure 7.1.



The binary tree representing the multiset:

{ `000` `000` `010` `011` `101` `110` `111` }.

The count at each node indicates the number of strings starting with the node's prefix: e.g. there are 7 strings starting with the empty string, 4 strings starting with `0`, 3 strings starting with `1`, and 2 strings starting with `01`.

The original multiset can be reconstructed from the tree.

Figure 7.1: Binary tree representing a multiset of seven fixed-length bit strings. Such a tree can be used, for example, to store a collection of SHA-1 hash sums in an order-invariant fashion.

To save space, nodes with zero counts may be omitted from the tree. For a multiset of fixed-length sequences, sequence termination is indicated by a leaf node, or a node that only has children with a count of zero. The sequence of branching decisions taken to reach any given node from the root is called the node's *prefix*. To recover the original multiset from the tree, it suffices to collect the prefix of each leaf node, including each prefix as many times as indicated by the leaf node's count.

A binary tree as described above is unique for any given collection of binary strings. The tree can be constructed incrementally, and supports addition, deletion and membership testing of sequences in $O(L)$ time, where L is the sequence length. Merging two trees can be done more efficiently than adding one tree's sequences to the other individually: the counts of nodes whose prefixes are equal can simply be added, and branches missing from one tree

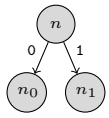
can be copied (or moved) from the other tree. Extracting N sequences from the tree, either lexicographically or uniformly at random, takes $O(L \cdot N)$ time.

7.2.2 Fixed-depth multiset tree compression algorithm

The previous section showed how a multiset of N binary sequences of fixed length L can be converted to a tree representation. This section derives exact conditional probability distributions for the node counts in the resulting tree, and shows how the tree can be compactly encoded with an arithmetic coder.

Suppose that N and L are known in advance. With the exception of the leaf nodes, the count n at any given node in the tree equals the sum of the counts of its children, i.e. $n = n_0 + n_1$. If the bits of each string are independently and identically distributed, the counts of the child nodes (conditional on their parent's count) jointly follow a binomial distribution:

$$\begin{aligned} n_1 &\sim \text{Binomial}(n,\ \theta) \\ n_0 &= n - n_1 \end{aligned} \qquad (7.1)$$



where θ is the probability of symbol `1`. If the symbols `0` and `1` are uniformly distributed (as is the case for SHA-1 sums), θ should be set to $\frac{1}{2}$. Given the parent count n, only one of the child counts needs to be communicated, as the other can be determined by subtraction from n.

Because all strings in the multiset have length L, all the leaf nodes in the tree are located at depth L, making it unnecessary to communicate which of the nodes are leaves.

If N and L are known, the tree can be communicated as follows: Traverse the tree, except for the leaf nodes, starting from the root (whose count N is already known). Encode one of child counts (e.g. n_1) using a binomial code (described in section 3.4.4), and recurse on all child nodes whose count is greater than zero. The parameters of the binomial code are the count of the parent, and the symbol bias θ, as shown in equation (7.1). The tree can be traversed in any order that visits parents before their children.

This encoding process is invertible, allowing perfect recovery of the tree. The same traversal order must be followed, and both N and L must be known (to recover the root node's count, and to determine which nodes are leaf nodes). Depending on the application, N or L can be transmitted first using an appropriate code over integers, such as those in sections 2.2 or 3.4.8. A concrete encoding and decoding procedure using a depth-first pre-order traversal of the tree can be found in code listing 7.1.

Application to SHA-1 sums. For a collection of N SHA-1 sums, the depth of the binary tree is $L = 160$, and the root node contains the integer N.

Coding algorithm for multisets of fixed-length sequences

ENCODING

Inputs: L, binary tree T

A. Encode N, the count of T's root node, using a code over positive integers.

B. Call `encode_node`(T).

DECODING

Input: L **Output:** binary tree T

A. Decode N, using the same code over positive integers.

B. Return $T \leftarrow$ `decode_node`(N, L).

subroutine `encode_node`(t):

If node t is a leaf:

1. Return.

Otherwise:

1. Let t_0 and t_1 denote the children of t, and n_0 and n_1 the children's counts.
2. Encode n_1 using a binomial code, as $n_1 \sim \text{Binomial}(n_0 + n_1,\ \theta)$.
3. If $n_0 > 0$, call `encode_node`(t_0).
4. If $n_1 > 0$, call `encode_node`(t_1).

subroutine `decode_node`(n, l):

If $l > 0$ then:

1. Decode n_1 using a binomial code, as $n_1 \sim \text{Binomial}(n,\ \theta)$.
2. Recover $n_0 \leftarrow (n - n_1)$.
3. If $n_0 > 0$, then: $t_0 \leftarrow$ `decode_node`$(n_0, l - 1)$.
4. If $n_1 > 0$, then: $t_1 \leftarrow$ `decode_node`$(n_1, l - 1)$.
5. Return a new tree node with count n and children t_0 and t_1.

Otherwise, return null.

Code listing 7.1: Coding algorithm for binary trees representing multisets of binary sequences of length L. The form and construction of the binary tree are described in section 7.2.1. Each tree node t contains an integer count n and two child pointers t_0 and t_1. The counts of the children are written n_0 and n_1. If n_0 and n_1 are zero, t is deemed to be a leaf, and vice versa. T denotes the tree's root node.

If the SHA-1 sums in the collection are random, then the distribution over the individual bits in each sequence is uniform, making a binomial code with bias $\theta = \frac{1}{2}$ an optimal choice. However, if we expect the collection to contain duplicate entries at a rate greater than chance, the distribution over the counts is no longer binomial with a fixed bias; in fact, the bias might then be different for each node in the tree. In such a case, a Beta-binomial code may be more appropriate, as it can learn the underlying symbol probability θ rather than assuming it to have a particular fixed value:

$$\begin{aligned} n_1 &\sim \text{BetaBin}(n, \alpha, \beta) \\ n_0 &= n - n_1 \end{aligned} \tag{7.2}$$

A Beta-binomial coding procedure was described in section 3.4.5. The tree coding method of code listing 7.1 can be modified to use a Beta-binomial code by replacing the encoding and decoding calls in the subroutine accordingly. In the experiments, the parameters of the Beta-binomial code were set to $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{2}$.

The practical performance of the algorithm on multisets of SHA-1 sums is shown in Figure 7.2. The multisets used in this experiment contain no duplicate hashes, so the compression achieved by the algorithm really results from exploiting the order-invariance of the multiset rather than any redundancy among the hashes. (When redundancy is introduced, the Beta-binomial code wins over the binomial code; but this scenario is not shown in the graph.)

7.3 Collections of binary sequences of arbitrary length

This section generalises the tree coding method to admit binary sequences of arbitrary length. Two approaches are considered for encoding the termination of sequences in the tree: the first approach covers collections of self-delimiting sequences, which allow the tree to be compressed without encoding additional information about termination. The second approach, for arbitrary sequences, assumes a *distribution* over sequence lengths and encodes sequence termination directly in the tree nodes. For either approach, the same binary tree structure is used as before, except that sequences stored in the tree can now have any length.

7.3.1 Compressing multisets of self-delimiting sequences

Self-delimiting sequences encode their own length, i.e. it can be determined from the sequence itself whether further symbols follow or the sequence has ended. Many existing compression algorithms produce self-delimiting sequences, e.g. the Huffman algorithm, codes for integers, and suitably defined arithmetic coding schemes. A multiset of such self-delimiting sequences

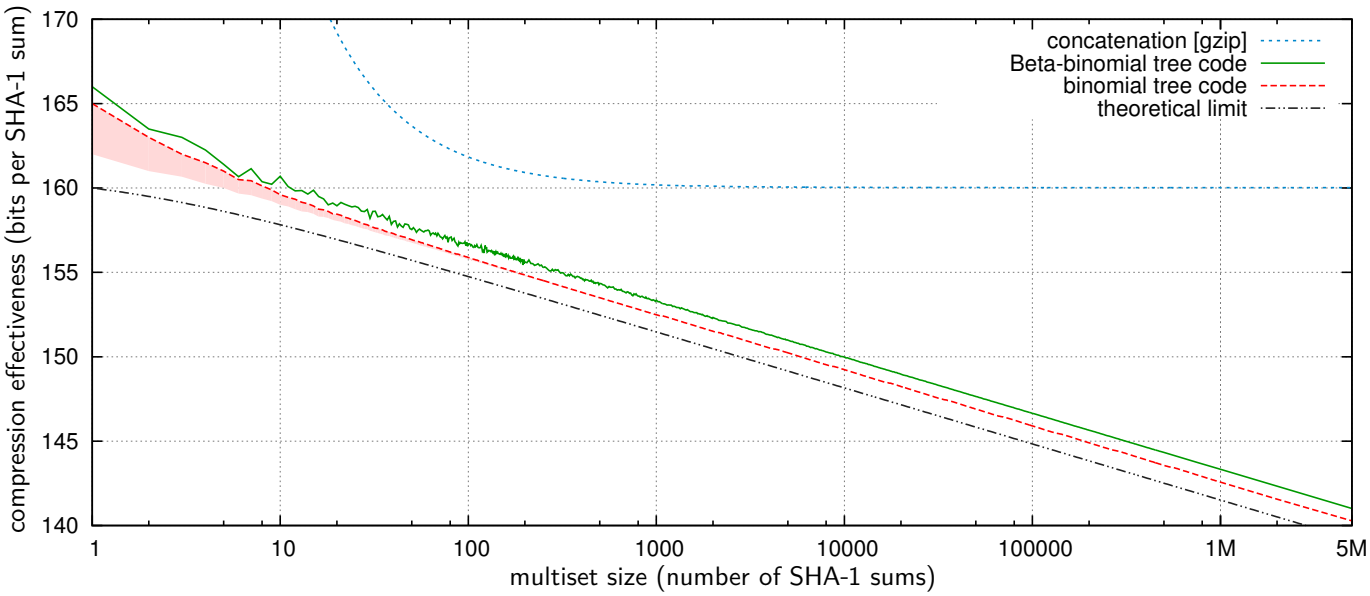


Figure 7.2: Practical compression performance of the fixed-depth multiset tree compressor on multisets of SHA-1 sums. For each position on the x-axis, N uniformly distributed 64-bit random numbers were generated and hashed with SHA-1; the resulting multiset of N SHA-1 sums was then compressed with each algorithm. The winning compression method is code listing 7.1 using a binomial code, where N itself is encoded with a Fibonacci code. The shaded region indicates the proportion of information used by the Fibonacci code. The theoretical limit is $160 - \frac{1}{N} \log_2 N!$ bits, assuming N is known to the receiver. For comparison, `gzip` was used to compress the concatenation of the N SHA-1 sums; reaching, as expected, a compression rate of 160 bits per SHA-1 sum.

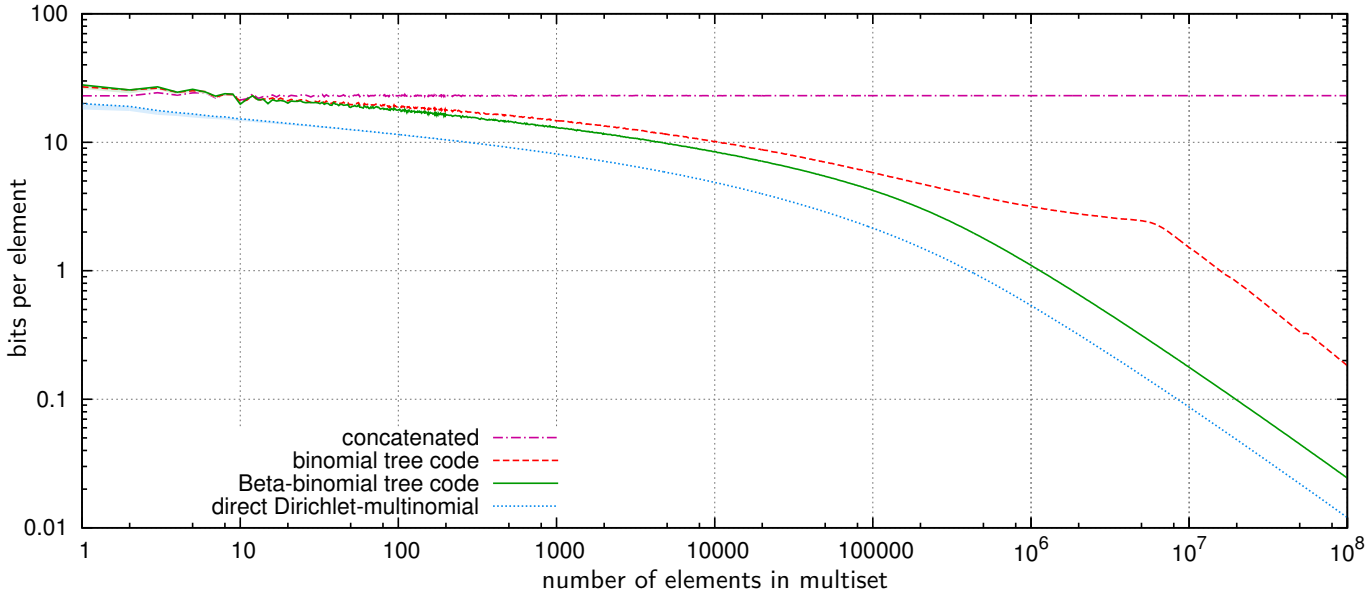


Figure 7.3: Experimental compression performance of various algorithms on multisets of self-delimiting sequences. For each position on the x-axis, a multiset of N self-delimiting sequences was generated by taking N uniformly distributed integers between 1 and 100 000 and encoding each with a Fibonacci code (described in section 2.2.4). The multiset of the resulting code words was then compressed with each algorithm.
The y-axis shows the compressed size in bits divided by N. The flat concatenation of the sequences in the multiset is included for reference (achieving zero compression). For comparison, the source multisets of integers (rather than the multisets of sequences) were compressed directly with a Dirichlet-multinomial multiset compressor, as described in section 5.5.1. The (barely visible) shaded regions indicate the amount of information taken up by the Fibonacci code to encode N itself.

has the property that for any two distinct sequences in the multiset, neither can be a prefix of the other.

Consider the tree corresponding to such a multiset of binary strings. Because of the prefix property, all sequences in the tree will terminate at leaf nodes, and the counters stored in child nodes always add up to the counter of the parent node. Consequently, the same compression technique can be used as for fixed-length sequences. Code listing 7.1 applies as before, with the exception that the end-of-string detector in the decoder must be modified to detect the end of each self-delimiting sequence.

Compressing arbitrary multisets. Consider a multiset $\mathcal{M}$ of objects from an arbitrary space $\mathcal{X}$, whose elements can be independently compressed to self-delimiting binary strings (and reconstructed from them). Any such multiset $\mathcal{M}$ can be losslessly and reversibly converted to a multiset $\mathcal{W}$ of self-delimiting sequences, and $\mathcal{W}$ can be compressed and decompressed with the tree coding method as described above.

Alternative. A random multiset $\mathcal{M}$ is most effectively compressed with a compression algorithm that exactly matches $\mathcal{M}$'s probability distribution; we'll call such an algorithm a

direct code for $\mathcal{M}$. When a direct code is not available or not practical, the indirect method of first mapping $\mathcal{M}$ to $\mathcal{W}$ might be a suitable alternative.

Experiment. Experimental results of this approach on random multisets of self-delimiting sequences are shown in Figure 7.3. Each multiset was generated by drawing N uniform random integers and converting these integers to self-delimiting sequences with a Fibonacci code (see section 2.2.4).[2] As expected, the Beta-binomial variant of the tree coder wins over the binomial variant when the number of repeated elements increases (as N exceeds 100 000). Note how the compression rate of the Beta-binomial tree code on the multisets of sequences follows the same trajectory as the (optimal) Dirichlet-multinomial encoding of the underlying multisets of integers (from which the multisets of sequences were produced). The gap is essentially the KL-divergence between the uniform distribution and the implicit distribution of the Fibonacci code (as plotted in Figure 2.1).

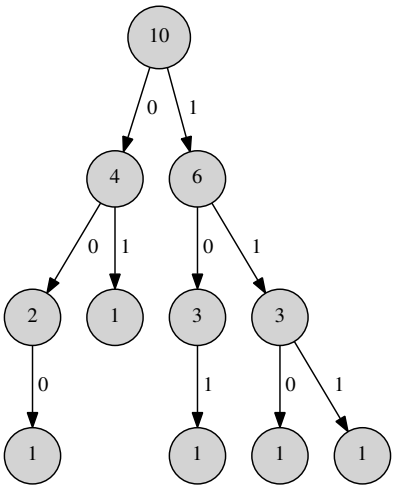
Self-delimiting sequences can also be produced with an arithmetic coder. In that case, arithmetic coding is used twice: first to compress each element to a binary string, and second to compress the resulting multiset of binary strings. The elements $x \in \mathcal{M}$ must be independently and identically distributed, as their binary string representations have to be decodable in any order (which makes it impossible to use a model that adapts from one x to the next).

This method may be useful when a direct multiset code (such as those described in section 5.5) cannot be constructed, or when the distribution over elements is not accessible. This tree coding technique can be used to structurally compress nearly any collection of elements whose members can be mapped to self-delimiting binary sequences. One of the drawbacks is that the one or two bit overheads incurred by the arithmetic coder when mapping the individual elements to binary sequences will accumulate, scaling linearly with the number of elements. However, for large enough multisets, the savings will eventually be worth it.

7.3.2 Encoding string termination via end-of-sequence markers

Consider now a multiset containing binary sequences of arbitrary length, whose sequences lack the property that their termination can be determined from a prefix. This is the most general case. In this scenario, it is possible for the multiset to contain strings where one is a prefix of the other, for example `01` and `011`. To encode such a multiset, string termination must be communicated explicitly for each string. Luckily, the existing tree structure can be used as before to store such multisets; the only difference is that the count of a node need not equal the sum of the counts of its children, as terminations may now occur at any node, not just at leaf nodes. Both child counts therefore need to be communicated. An example of such a tree is shown in Figure 7.4.

[2] The Fibonacci code was chosen for elegance. However, any code over integers could be used, e.g. an exponential Golomb code (Teuhola, 1978) or the ω-code by Elias (1975).



The binary tree representing the multiset:

{ 0, 00, 000, 01, 10, 10, 101, 11, 110, 111 }.

The count at each node indicates the number of strings starting with that node's prefix. For example, there are 10 strings starting with the empty string, 4 starting with 0, and 6 starting with 1, etc.
Out of the four strings starting with 0, two continue with 0, one continues with 1, and one reached its termination.
If the root node's count were 12 rather than 10, the multiset would include two empty strings as well.

Figure 7.4: Binary tree representing a multiset of binary sequences of finite but arbitrary length. This tree follows the same basic structure as the tree in Figure 7.1, but admits sequences of variable length. The tree representation is unique for each multiset.

The counter n stored in each node still indicates the number of sequences in the collection that start with that node's prefix. The number of terminations n_τ at any given node equals the difference of the node's total count n and the sum of its child counts n_0 and n_1.

Suppose that the total number $N = |\mathcal{W}|$ of sequences in the multiset $\mathcal{W}$ is distributed according to some distribution D over positive integers, and that the length of each sequence $w_n \in \mathcal{W}$ is distributed according to some distribution L. Given D and L, a Shannon-optimal compression algorithm for the multiset $\mathcal{W}$ can be derived as follows.

Form the tree representation of $\mathcal{W}$, following the construction described in the previous section. The count of the root node can be communicated using an appropriate code for D. Each node in the tree has a count n, child counts n_0 and n_1, and an implicit termination count n_τ fulfilling $n = n_0 + n_1 + n_\tau$. Assuming that the bits at the same position of each sequence are independently and identically distributed, the values of n_0, n_1 and n_τ are multinomially distributed (given n).

The parameters of this multinomial distribution can be determined from L as follows: The n sequences described by the current node have a minimum length of d, where d is the node's depth in the tree (the root node is located at depth 0). Out of these n sequences, n_0 continue with symbol 0, n_1 continue with symbol 1, and n_τ terminate here. As was shown in section 5.7.2, the probability of a sequence which has at least d symbols to have no more than d symbols is given by a Bernoulli distribution with bias $\theta_\tau(d)$, where:

$$\theta_\tau(d) \;:=\; \frac{L(d)}{1 - \sum_{k<d} L(k)} \tag{7.3}$$

Consequently, the number of terminations n_τ at depth d (out of n possible sequences) is

binomially distributed with:

$$n_{\mathtt{T}} \sim \text{Binomial}(n,\ \theta_{\mathtt{T}}) \tag{7.4}$$

Writing $\theta_{\mathtt{T}}$ for the probability of termination at the local node, and $\theta_{\mathtt{1}}$ and $\theta_{\mathtt{0}}$ for the occurrence probabilities of `1` and `0`, the joint distribution over $n_{\mathtt{0}}$, $n_{\mathtt{1}}$ and $n_{\mathtt{T}}$ can be written as follows:

$$(n_{\mathtt{T}}, n_{\mathtt{0}}, n_{\mathtt{1}}) \sim \text{Mult}\Big(n,\ \left(\theta_{\mathtt{T}},\ \theta_{\mathtt{0}}\left(1-\theta_{\mathtt{T}}\right),\ \theta_{\mathtt{1}}\left(1-\theta_{\mathtt{T}}\right)\right)\Big) \tag{7.5}$$

where $\theta_{\mathtt{1}} = 1-\theta_{\mathtt{0}}$. The encoding procedure for this tree needs to encode a ternary (rather than binary) choice, but the basic principle of operation remains the same. Code listing 7.1 can be modified to encode $(n_{\mathtt{T}}, n_{\mathtt{0}}, n_{\mathtt{1}})$ using a multinomial code, e.g. as described in section 3.4.6.

Note that, as described above, $\theta_{\mathtt{T}}$ is a function of the length distribution L and the current node depth d. In principle, it is possible to use a conditional length distribution that depends on the prefix of the node, as the node's prefix is available to both the encoder and the decoder. Similarly, $\theta_{\mathtt{0}}$ and $\theta_{\mathtt{1}}$ could in principle be functions of depth or prefix.

7.4 Conclusions

This chapter proposed a novel and simple data compression algorithm for sets and multisets of sequences, and illustrated its use on collections of cryptographic hash sums, and on multisets of Fibonacci-encoded integers. The approach of this method is based on the general principle that one should encode a permutation-invariant representation of the data, in this case a tree, with a code that matches the probability distribution induced by the data's generative process. When the sequences in the source multiset are *iid* and random from a known distribution, the tree is most effectively compressed using code listing 7.1 with a binomial code; otherwise, a Beta-binomial code can be used instead. The Beta-binomial code is universal in that it learns the symbol distribution of the sequences in the multiset (even for symbol distributions that are position- or prefix-dependent).

One might regard the coding algorithms presented in this chapter as either lossless compression for sets and multisets, or lossy compression methods for lists: when the order of a list of elements isn't important, bandwidth can be saved.

Note. A version of this chapter was presented at the Data Compression Conference in 2014.

Chapter 8

Cost-sensitive compression and adversarial sequences

This chapter considers the following two ideas, and gives examples of their applications:

1. Lossless compression algorithms are designed to produce compact representations of input objects, minimising the transmission cost of the output sequence. If the symbols in the output alphabet have equal transmission costs, then the task of minimising the cost is equal to the task of minimising the number of output symbols, and the optimal distribution over those output symbols (i.e. the *target distribution*) is uniform. However, if the output symbols do *not* have equal transmission costs (such as in Morse code), the optimal target distribution is *not* uniform. Arithmetic coding can be generalised to compress to non-uniform target distributions, and section 8.1 shows how to compute the optimal target distribution for a given alphabet and associated transmission costs.

2. Any compression algorithm necessarily defines a probability distribution over its input objects. This distribution can (in principle) be used to generate samples, or to compute worst-case input objects for which the compression algorithm produces the *longest* possible output sequences. For context-sensitive compressors (such as PPM, BPPM or LZW), section 8.3 discusses *adversarial* input sequences that are constructed by greedily choosing a least predicted symbol for each position in the sequence. While these greedy adversarial sequences are not necessarily worst-case inputs to the compressor for which they were constructed, they are worse than random sequences and have some surprising properties that are illustrated at the end of this chapter.

8.1 Cost-sensitive compression

Most compression algorithms are designed to compress data into a sequence of uniformly distributed output symbols. When the costs of transmitting each output symbol are equal, the uniform target distribution minimises the expected transmission cost of the compressed output. This section considers what happens when the transmission costs of the output symbols are *not* equal, and shows how to compute the optimal target distribution given the transmission costs of the output symbols.

The arithmetic coding algorithm can be amended to directly produce output symbols from a different target distribution. Alternatively, a 'two-pass approach' can be used by applying the standard arithmetic coding algorithm twice: first to compress the input object into a sequence of uniformly distributed bits, and second to decompress those bits from the first pass into symbols of the chosen target distribution. An example application is given in section 8.1.2.

8.1.1 Deriving the optimal target distribution

Given an output alphabet $\mathcal{Y}$ and associated symbol costs c_y, let's derive the optimal output symbol probabilities p_y such that the rate of information transmission (per unit of transmission cost) is maximised. The rate is equal to the average information content per symbol H divided by the average symbol cost C:

$$R = \frac{\text{avg. information per symbol}}{\text{avg. cost per symbol}} = \frac{H}{C} = \frac{\sum p_y \log \frac{1}{p_y}}{\sum p_y c_y} \tag{8.1}$$

We wish to find the values of the p_y that maximise R subject to the constraint that the p_y sum to unity. This maximum can be found e.g. using a Lagrange multiplier λ that enforces the constraint:

$$G(\boldsymbol{p}) := R(\boldsymbol{p}) + \lambda \left(\left(\sum_{y \in \mathcal{Y}} p_y \right) - 1 \right) \tag{8.2}$$

At the location $\boldsymbol{p}$ where $R(\boldsymbol{p})$ has its maximum, the partial derivative with respect to p_y must be zero:

$$\frac{\partial G}{\partial p_y} = \frac{\partial}{\partial p_y} \left(\frac{H}{C} + \lambda \sum p_y \right) = 0 \tag{8.3}$$

The partial derivative is equal to:

$$\frac{\partial G}{\partial p_y} = \frac{\partial}{\partial p_y} \left(\frac{H}{C} + \lambda \sum p_y \right) \tag{8.4}$$

$$= \frac{-H}{C^2} \frac{\partial C}{\partial p_y} + \frac{1}{C} \frac{\partial H}{\partial p_y} + \lambda \tag{8.5}$$

$$= \frac{-H}{C^2}c_y + \frac{1}{C}\left(\log\frac{1}{p_y} - 1\right) + \lambda \tag{8.6}$$

And solving for p_y gives:

$$\log\frac{1}{p_y} = \frac{H}{C}c_y + 1 - \lambda C \tag{8.7}$$

$$p_y = \exp\left(-\frac{H}{C}c_y\right) \cdot \exp\left(1 - \lambda C\right). \tag{8.8}$$

Finding λ to make the p_y sum to unity is equivalent to finding the normalising constant Z:

$$p_y = \frac{1}{Z}\exp\left(-\frac{H}{C}c_y\right) \tag{8.9}$$

Unfortunately, equation (8.9) is recursive as both H and L contain p_y, but numerical methods can typically be used to determine p_y.

Example. Consider the case of a binary symbol alphabet {■, ▬} where the transmission cost of symbol ▬ is twice that of symbol ■. Clearly, a uniform distribution over {■, ▬} isn't optimal. Choosing $\Pr(■) = \frac{2}{3}$ and $\Pr(▬) = \frac{1}{3}$ may seem tempting, as this distribution spends (on average) equal expense on each symbol type. However, this distribution does not achieve the optimal information rate, either. The solution to equation (8.9) reveals the optimal distribution to be:

$$\begin{aligned}\Pr(■) &= \frac{\sqrt{5}-1}{2} = \frac{1}{\varphi} &&= 0.618033988749895\ldots \\ \Pr(▬) &= \frac{3-\sqrt{5}}{2} = \frac{\varphi-1}{\varphi} &&= 0.381966011250105\ldots\end{aligned} \tag{8.10}$$

where $\varphi = 1.618033988749\ldots$ is the golden ratio. A graph of the communication rate as a function of $\Pr(■)$ can be found in Figure 8.1 (on page 160). The optimal rate R, achieved for $\Pr(■) = \frac{1}{\varphi}$, equals $\log_2\varphi$ bits per unit cost (where $\log_2\varphi \approx 0.694241913630$).

8.1.2 Optimising Morse code

A example of a coding system whose symbols have unequal transmission costs is the Morse code, which predates the foundational work on information theory and coding by Shannon (1948) by more than a century. Morse code was designed to be used by human operators for long distance communication.

Morse code was first described by Samuel Morse (1840) in his patent application for the "American Electro-Magnetic Telegraph", and most likely developed in close collaboration with Alfred Vail, who described it in a book on the same subject (Vail, 1845). The original code employed several different signal durations, and was changed significantly by Carl Au-

A	▪▬	8	J	▪▬▬▬	16	S	▪▪▪	8
B	▬▪▪▪	12	K	▬▪▬	12	T	▬	6
C	▬▪▬▪	14	L	▪▬▪▪	12	U	▪▪▬	10
D	▬▪▪	10	M	▬▬	10	V	▪▪▪▬	12
E	▪	4	N	▬▪	8	W	▪▬▬	12
F	▪▪▬▪	12	O	▬▬▬	14	X	▬▪▪▬	14
G	▬▬▪	12	P	▪▬▬▪	14	Y	▬▪▬▬	16
H	▪▪▪▪	10	Q	▬▬▪▬	16	Z	▬▬▪▪	14
I	▪▪	6	R	▪▬▪	10	␣		4

Table 8.1: International Morse Code, as specified by ITU-R (2009). The code words are composed of two signals {▪, ▬}, separated by pauses. Signal ▪ has length 1, and ▬ has length 3. Pauses between signals (in the same letter) have length 1. Pauses between letters have length 3. Pauses between words have length 7. Example: "▬▪▬ ▪ ▪ ▪▬▬▪ ▪▪ ▬ ▪▪▪ ▪ ▬▪▬▪ ▪▬▪ ▪ ▬".
The numbers in the table indicate the total length of each code word (including pauses).

gust von Steinheil and Friedrich Clemens Gerke for use on the European continent (Gerke, 1851). The combination of Gerke's three-signal and Steinheil's two-signal code resulted in the International Morse Code which is still in global use today. International Morse Code was standardised multiple times (and occasionally updated), most recently by ITU-R (2009).

A table of the code (excluding numerals and punctuation) is shown in Table 8.1. The code defines two signals (▪ and ▬), where the long signal ▬ is three times as long as the short signal ▪. The Morse code table maps each letter of the Latin alphabet to short sequences of signals separated by pauses; somewhat reminiscent of a symbol code (as described in section 2.1).

If a computer system (rather than a human) were to communicate using Morse code signals, it would be sensible to abandon the code word table entirely. Using a probabilistic language model and arithmetic coding will make much better use of the bandwidth, directly producing a stream of ▪:s and ▬:s (separated by pauses).

The intersignal pauses, as defined by International Morse Code, have the same duration as the short signal ▪. Let's simplify by assuming that only these short pauses are used. Writing "▪▫" for "▪ + pause", and "▬▫" for "▬ + pause", what probability distribution over {▪▫, ▬▫} maximises the rate of communication in such a system? Adding up, the duration of ▬▫ is exactly twice the duration of ▪▫. Consequently, the output distribution which achieves the optimal communication rate follows the golden ratio, as given in equation (8.10).

As it may seem a bit optimistic to expect human operators to run arithmetic coding in their heads when communicating in Morse signals, let's briefly revisit the Morse code table and reflect on its effectiveness for communicating human text. The lengths of the Morse

A	▪▪	6	−	J	▪▬▬▪▪	14	−	S	▪▪▬	10	+
B	▬▪▪▪	14	+	K	▬▪▬▪	14	+	T	▬	6	
C	▬▪▬▪	12	−	L	▬▪▪	10	−	U	▪▪▬▪▪	12	+
D	▬▪▪	10		M	▬▬▪	12	+	V	▪▪▬▪	14	+
E	▪	4		N	▬▪	8		W	▪▬▬	12	
F	▪▪▬▪	12		O	▬▬▬	8	≡	X	▬▪▪▬	14	
G	▬▬▪	12		P	▬▪▪▪	12	−	Y	▪▬▬	14	−
H	▪▪▪▪	10		Q	▬▬▪▬	14	−	Z	▬▬▪▪	14	
I	▪▪▪	8	+	R	▪▬▪	10		␣		4	

Table 8.2: A new Morse code for English: allocation of code words based on contemporary English letter frequencies by Norvig (2013), as shown in Figure 4.1 (on page 63). Signal durations and pauses are identical to International Morse Code (see Table 8.1). The existing code words for `DEFGHNRTWXZ` were kept, as they were already optimal. Code words for `ACJLPQY` were shortened by 2 units, and code words for `BIKMSUV` were lengthened by 2 units. The code word for `O` was shortened by 6 units. The code word ▪▪▪▪▪ was avoided for practical reasons (compatibility with existing code words for digits and punctuation marks), otherwise the code is optimal. Example:
“▪▪▪▪ ▪ ▬▬ ▬▬ ▪▬ ▬▪ ▪ ▪▬▬ ▪▪▪▬ ▪▬ ▬▪▪ ▪ ▬ ▪▪ ▪▬▬▪ ▬▬ ▪”

code words should be chosen such that the expected message duration is minimised: the shortest code word should go to the most frequent letter. Since the code words are separated by a forced interletter pause of length 3, there is no need for the code to have the prefix property. An optimal mapping from letters to code words can therefore be produced simply by generating all Morse code words in order of increasing length, and assigning each to the next most frequent letter until all letters have been allocated.[1]

A revised Morse code table for English, constructed as described above, is presented in Table 8.2. A comparison of the communication rates of traditional and new Morse code is given in Table 8.3. The table also shows the communication rates of a PPM language model whose arithmetic coder directly produces a sequence of {▪, ▬} signals.

Aside. It may be surprising that International Morse Code (as shown in Table 8.1) assigns a relatively long code word (▬▬▬) to the letter `O`, suggesting a much lower occurrence probability than typically found in English. The original code words might have been constructed for German (not English) letter frequencies, where `O` occurs much more rarely.

[1]The number of possible Morse code words for any fixed length is given by a Fibonacci number. A closed form for Fibonacci numbers also involves the golden ratio φ:

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n = \frac{1}{\sqrt{5}}\left(\varphi^n - (-\varphi)^{-n}\right) \tag{8.11}$$

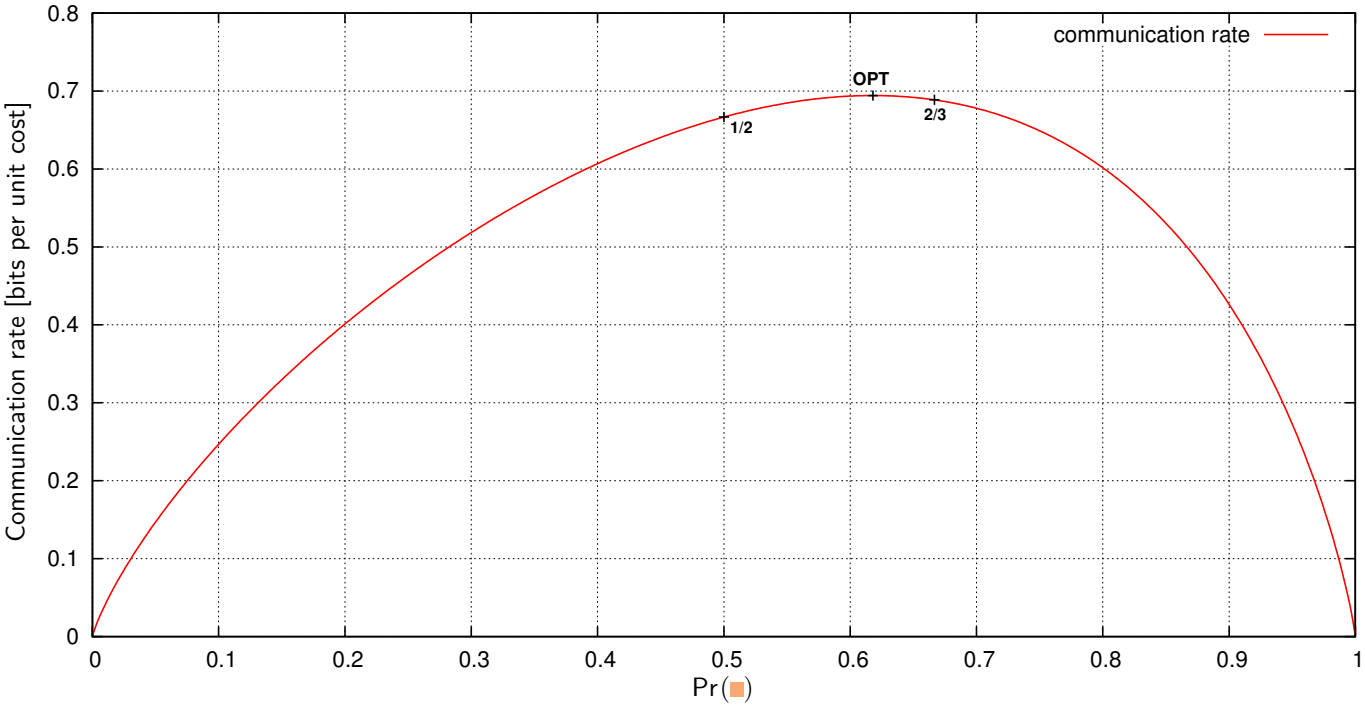


Figure 8.1: Communication rate of an output symbol alphabet {■, ▬} with unequal symbol durations, as a function of Pr(■) = 1 − Pr(▬). The duration of ▬ is twice that of ■. The optimum is located at $\frac{1}{\varphi}$, the inverse of the golden ratio.

Static Method			**units**	**u / sym**	**time to transmit** (at 40 u/s)
Old Morse code (Table 8.1)			37345863	7.612	258 hours 54 mins
New Morse code (Table 8.2)			35758353	7.288	248 hours 17 mins
Method	Pr(▪)	Pr(▬)	**units**	**u / byte**	**time to transmit** (at 40 u/s)
PPMD5	1/2	1/2	32382056	6.129	224 hours 53 mins
PPMD5	2/3	1/3	31340450	5.931	217 hours 39 mins
PPMD5	φ^{-1}	$1-\varphi^{-1}$	31090434	5.884	215 hours 55 mins

Table 8.3: Comparison of different Morse coding systems. The methods were compared on `shakespeare.txt`, the concatenated works of Shakespeare. The table shows each method's communication rate in units per symbol, where one time unit equals the duration of one ▪, as described in Table 8.1.
For the static methods (old and new Morse code), the input sequence was made case insensitive, stripped of all punctuation and newlines, and white space was folded into single spaces. For the adaptive methods, the original (unsimplified) sequence was used. The adaptive methods used PPM (depth 5, escape method D) to compress the input sequence, differing only in the choice of emission probabilities Pr(▪), Pr(▬).
For illustration, the final column shows the time it would take to transmit the encoded message at 40 units per second.

8.2 Compression and sampling

An *exact sampling algorithm* for a probability distribution D produces independent samples from D, given a source of randomness. Efficient exact sampling algorithms exist for many basic probability distributions, see e.g. Devroye (1986) for descriptions of such algorithms. Exact sampling methods can be derived even for some complicated distributions, a notable family of methods being *coupling from the past* (Propp and Wilson, 1996, 1997). In general, however, it is rather difficult to design exact sampling algorithms with reasonable resource costs, especially for posterior distributions from complex probabilistic models. Approximate sampling algorithms are therefore often used in practice.

It turns out that any probabilistic model that can be used with an arithmetic coder can also be used to produce exact samples. Making an *arithmetic decoder* decompress a stream of uniformly distributed random bits results in an exact sample from the probabilistic model it is interfaced to. The exact sample is produced using only the minimal number of random bits required (Cover and Thomas, 2006, section 5.11). As this process is invertible, the matching *arithmetic encoder* can reconstruct, for any given element of the sample space, the exact sequence of (originally random) bits that the decompressor used for sampling it.

Note that this "sampling by decompression" scheme is only a valid procedure if the compressor defines a bijective mapping between the input objects and output sequences; otherwise, decompression of a random sequence can produce biased samples or undefined behaviour.

Samples from a probabilistic model can be useful for computing approximate expectations, or for visualising and understanding some of the model's properties.

Table 8.4 shows sequences that were sampled from a fixed-depth BPPM after being trained on *Alice's Adventures in Wonderland*.

8.3 Worst case compression and adversarial samples

No lossless compression algorithm can expect to compress a perfectly random input sequence. In particular, for an input sequence that is constructed by drawing symbols uniformly at random from an alphabet $\mathcal{X}$, the best possible compression effectiveness that can be expected is $\log_2|\mathcal{X}|$ bits per symbol, the entropy of the uniform distribution over $\mathcal{X}$. (In fact, a typical adaptive compressor will do slightly worse, as it has to learn the distribution and also encode the length of the sequence.)

Figure 8.4 on page 168 (also Figure 2.3b on page 36) shows the compression effectiveness of various compression algorithms on a pseudo-random input sequence, as a function of input length. This random input sequence (Seq. I) was created by making independent draws from a uniform distribution over an alphabet of 64 symbols (out of 256 possible symbols), using

$D=0$:	Compress esset' k@ rgeO ,n d uea pmitilte@' eur r,'epeei iisosoa@i -a.e sei h...
$D=1$:	Compress helllan helithed t fithorye, t t sats cut ore mphot Tigay a way aw t...
$D=2$:	Compress ne over-@ that she wasid Theen ance de and said som thiteavery@eve has...
$D=3$:	Compress sizes!'@@@ 'One for hairst thing to@here I once fixtig curiously; 'lipp...
$D=4$:	Compress soon at or you nonsense!' Becaused soon as if a disapples well my dears...
$D=5$:	Compress his head. 'Then your Majesty,' the sented at once to usurprised to him,...
$D=6$:	Compress him), whatever,' said the small@passage, I'll each other question is, Wh...
$D=7$:	Compress him! Pinch@him! Pinch@himself from beginning again?' the Hatter; 'What...
$D=8$:	Compress him! Pinch@him! Off with his shoulder with all sorts of the other. 'I...
$D=9$:	Compress him! Pinch@him! Off with his nose@ Trims his belt and@day! Why, I...
$D=5$:	Never this head pretending, I gloves when the pictured to the@reply.@@ 'Why,' sa...
$D=5$:	Never were out as he went branches on whis which way of edition?'@@ 'It is all f...
$D=5$:	Never way of sitting to do wish I have and@she was very sudden change, as@she hea...
$D=5$:	Never you've no archbishop could be sure; seemed to@ cur, "You our...
$D=5$:	Never saw he@jumped upon it@would terms without@perhaps out at all, while them, t...
$D=5$:	Never crone with curious dropped in croquet.'@@ She@could YOU with his eyes a Ha...
$D=5$:	Never could have ground,@'and not remarked: 'UNimportant howling--@people joys...
$D=5$:	Never coming@serpent, afore.'@@ Alice in at here. Alice replied right size; but...
$D=5$:	Never complainly now the seated the found the blows slivery: other Williamond...
$D=5$:	Never comprove I didn't@believe to@yet hastily replied: 'how am I to get use...
$D=5$:	Never compressinging that Alice asked Alice with you join to beginning but I ment...
$D=5$:	Never compressed to be a soldiers and shut his began his pocket!'@@ Alice did th...
$D=5$:	Never compressed, and@serpent in.@@ 'We indeed!'@@ 'They all remark, ared, 'tha...
$D=5$:	Never compress it led in a minutes nose, in a long nonsense!' (W-in March Hare wa...
$D=5$:	Never compress was there's no used to the@triump!'@@ 'Repeat@sorts@of life, and,...
$D=5$:	Never compress with his time whether laddressed, 'If you like it sort in a great...
$D=5$:	Never compress with the Mouse@I'm ceived up on to put me, but never the grow at t...
$D=5$:	Never compress with the March Hare tarts of March Hare were says it?' he song abo...
$D=5$:	Never compress without key; and graved):@ * For supp...
$D=5$:	Never compress without kindly enjoy@ Waition a sharply wish I hard if it...
$D=5$:	Never compress without knocking, she little watch an@remember@even was this voice...
$D=5$:	Never compress without every,@cause@I'm afraid in asking, and green...
$D=5$:	Never compress without again, they're done@inch; and the right used at him with o...
$D=5$:	Never compress without a moment by the English,' the Rable door, and she did not,...

Table 8.4: String continuations sampled from BPPM ($\alpha = 0$, $\beta = \frac{1}{2}$) of various context depths D, pre-trained on *Alice in Wonderland* by Lewis Carroll (1865), file `alice29.txt` of the Canterbury Corpus. The shaded part of each line marks the string that BPPM was asked to continue.
Each continuation was produced by incrementally sampling from BPPM's sequence of predictive symbol distributions (conditional on the preceding symbols). Newline characters are represented by @-symbols. To make the above samples reproduceable, the seed of the pseudo-random number generator was initialised to 42 for each sequence.

a pseudo-random number generator with a fixed random seed. As one would hope, most compressors approach a compression rate of 6 bps.[2]

Interestingly, random input sequences aren't necessarily the worst-case input of a compression algorithm. Compression methods can, in addition to obtaining exact samples of their distributions, be used to generate non-typical "worst-case" samples: we can inspect the compressor's distribution and pick an element that's least predicted. In practice, a generic search for worst-case inputs can be prohibitively expensive, especially for complicated adaptive sequence compressors.

Instead of searching for worst-case sequences, we can linearly construct greedy *adversarial sequences* that are guaranteed to be worse than (or as bad as) sequences of independent and uniformly distributed random symbols. The following section describes the construction of such sequences and documents some of their curious properties.

8.3.1 Adversarial sequences

Adaptive sequence compression algorithms like PPM or the Sequence Memoizer define a series of dependent predictive probability distributions. These distributions can be used to play an adversarial game where, for each predictive distribution, an element with least probability mass is picked. If there are several such elements, we could pick according to some deterministic rule (e.g. in the order of the alphabet) or uniformly at random. Both of these policies result in interesting sequences.

Note that the greedy construction described here might not necessarily produce an overall worst-case input sequence; however, the sequential adversarial selection of input symbols is (by construction) worse for the victim than picking symbols uniformly at random. This effect is shown in Figure 8.2.

Finite depth Markov models

Let's look at some adversarial sequences for simple sequence models, using the binary input alphabet $\{0, 1\}$, where both 0 and 1 have equal probability mass a priori.

To start, consider an adaptive zero-context symbol model, such as the Dirichlet process compressor from section 4.2.2, or a depth one BPPM (with $D = 1$, $\beta = 0$ and $\alpha > 0$). The resulting adversarial sequence (with alphabetic precedence rule) is:

0101010101010101010101010101010101010101010101010101010101010101...

[2]As the sequence is pseudo-random, it is technically deterministic. A strong AI compressor could, in principle, crack the random seed and compress the sequence much better than playing against chance. No such compressor has ever been built, and a general non-approximate AI compressor is uncomputable (Hutter, 2004).

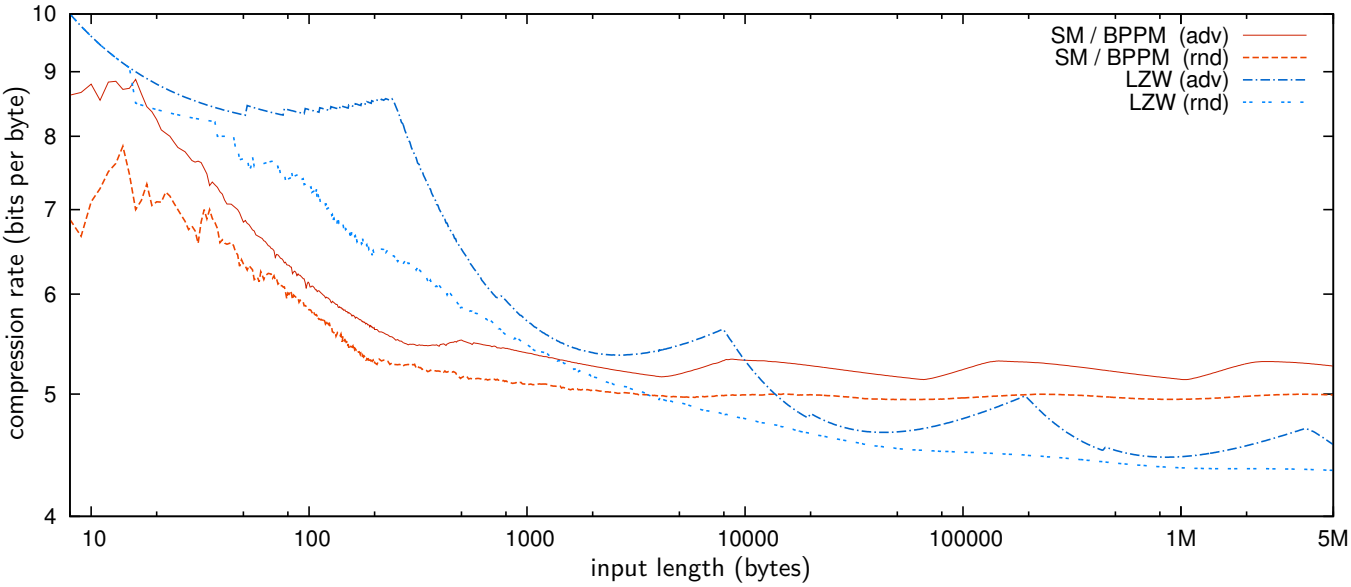


Figure 8.2: The compression effectiveness of two algorithms (BPPM and LZW), each on a random sequence (Seq. XII, uniformly distributed symbols from the set {0–9, A–F}), and on an adversarial sequence (Seq. X for BPPM, and Seq. XX for LZW).

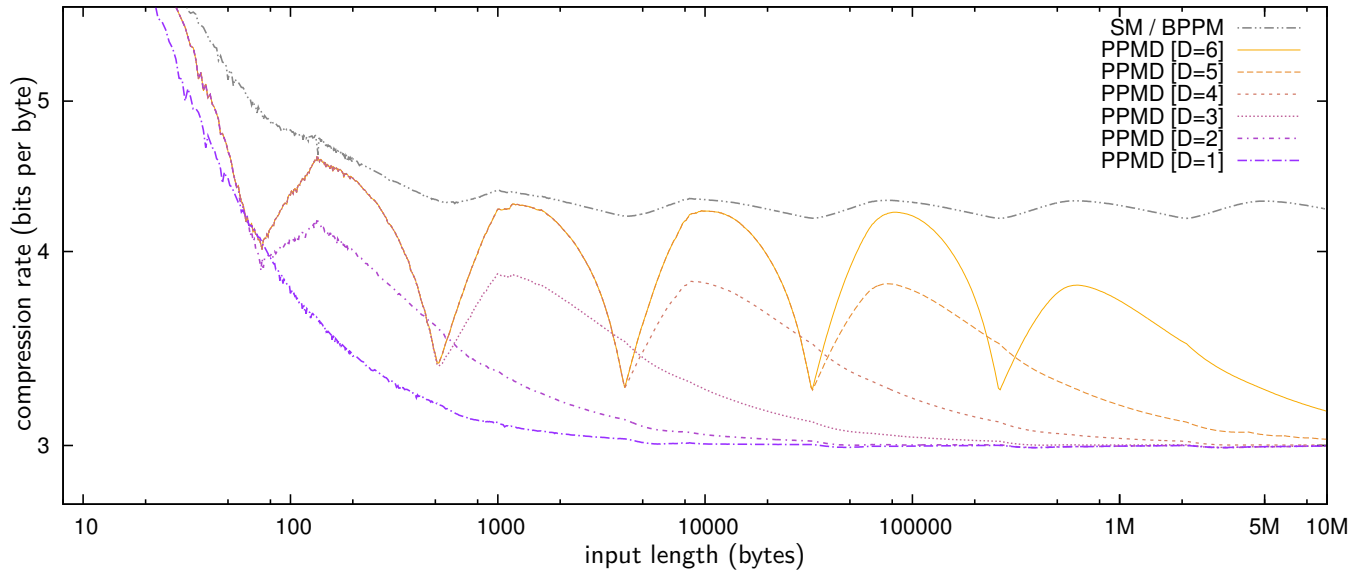


Figure 8.3: The compression rates of PPMD (for various settings of the context depth D) on Seq. XIV, an alphabetic adversarial sequence for BPPM / the Sequence Memoizer using a subset of 8 symbols {0–7} from the byte alphabet. The adversarial sequences makes PPM's compression rate trace out $D-1$ hills, before making a final approach to 3 bits per symbol. A PPM with unbounded depth (such as PPM*) produces hills ad infinitum.

The above sequence arises as follows: the predictive distribution for the first symbol has $\Pr(\texttt{0}) = \Pr(\texttt{1}) = \frac{1}{2}$, so `0` is chosen because it comes first in the alphabet. The predictive distribution for the second symbol (conditional on the first) is:

$$\Pr_{\texttt{0}}(\texttt{0}) = \frac{1+\frac{\alpha}{2}}{1+\alpha} \quad \text{and} \quad \Pr_{\texttt{0}}(\texttt{1}) = \frac{\frac{\alpha}{2}}{1+\alpha}. \tag{8.12}$$

The adversary therefore picks `1`, the less probable, for the second symbol. For the third symbol, both symbols are equally probable again: $\Pr_{\texttt{01}}(\texttt{0}) = \Pr_{\texttt{01}}(\texttt{1}) = \frac{1}{2}$ and the alphabetic rule again chooses `0`. The sequence continues with `01` repeating forever: at odd positions, each symbol has probability $\frac{1}{2}$ and is chosen by alphabetic precedence, and the symbols at even positions are necessarily the opposite of the immediately preceding symbol.

Consider now the corresponding adversarial sequence for BPPM with depth 1:

01001 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100...

This sequence generates all bigrams, with overlapping digits, and ends up repeating `1100` ad infinitum. A slightly puzzling property might be the irregularity at the beginning of the sequence: after `010011`, a `1` is picked rather than a `0`, although both `0` and `1` have occurred three times each. The reason why BPPM gives lower probability to `1` in this context is because of the shallow update (1TPD) rule: `0` creates a new table for three contexts (ε, `0` and `1`), whereas `1` creates a new table for only two contexts.

Here is the adversarial sequence for BPPM with depth 2:

010011101110 00010111 00010111 00010111 00010111 00010111 00010111...

Much the same thing happens here: all bigrams and trigrams are generated, and the sequence eventually becomes periodic. A similar pattern emerges for the adversarial sequence of context depth 3:

0100111011000010100001 1111001011010000 1111001011010000 1111001011010000

These characteristics are not limited to the binary alphabet. For example, here is the adversary sequence for BPPM depth 1 on the ternary alphabet $\{\texttt{a}, \texttt{b}, \texttt{c}\}$:

abcacbaabbccb baccaabbc baccaabbc baccaabbc baccaabbc baccaabbc...

The cyclic subsequences (`01`, `1100`, `00010111`, `baccaabbc`, etc) which appear in the periodic part of the adversarial sequence are *de Bruijn sequences* (de Bruijn, 1946). A K-ary de Bruijn sequence is a cyclic arrangement of symbols such that every possible subsequence of length K occurs exactly once. The adversarial sequences for BPPM of any finite context depth (and any finite alphabet) eventually settle in a de Bruijn cycle that repeats indefinitely.

Infinite-depth Markov models

In the case of a Markov model with unbounded depth, adversarial sequences do not have periodic fixed-length repetitions. For example, the following sequence (Seq. XVIII) is adversarial for the Sequence Memoizer, with all discount parameters set to $\beta = \frac{1}{2}$:

```
0100111011000010100011010111110010000001111010010110111000100110011
1111101111000001000101110101011001010100110110100001100011100110000
0000101100010101110010111101101100110100111100111000011101000100001
0010010100100011111010110101000001101111110001100100111110000100000
1010101011110010100001011111111100110111011100111101110100110001000
1100001101000111011111011001000100101110000001001110010010000111111
0100000001100110010110011101011101101011000001110001111000101101001
0011010101000101001110001010000001010010101101101111010100101111101
0011101001000001111100111111011100001011011000110110000001101010011 1
1110010110101110001101001010001001111010111101000110011100111001110...
```

The sequence continues indefinitely but has no periodic repetitions. Note that the demand for additional context depth grows very slowly: The adversarial sequence generator will generate as many context misses as possible, exhausting each depth completely before moving on to the next. For example, the adversarial sequence of length 100 000 for the Sequence Memoizer's "batch Deplump (UKN)" method, on an alphabet with 64 symbols, can be generated much more cheaply using BPPM with fixed depth $D = 3$. Up to this length, the sequences produced by both algorithms are identical.

8.3.2 Some results

As an exploratory piece of research, a collection of adversarial and random sequences were generated for various alphabet sizes (each a subset of the 256 symbols in the byte alphabet), and compressed with different algorithms.

The somewhat striking results of this experiment are documented in a series of graphs over the next few pages (166–178). Each graph shows the compression rates (in bits per byte) of various algorithms on one particular sequence, as a function of input length. Each sequence is described below the plot, indicating its type, symbol set, and the first few symbols in the sequence. The sequences are also summarised in Table 8.6.

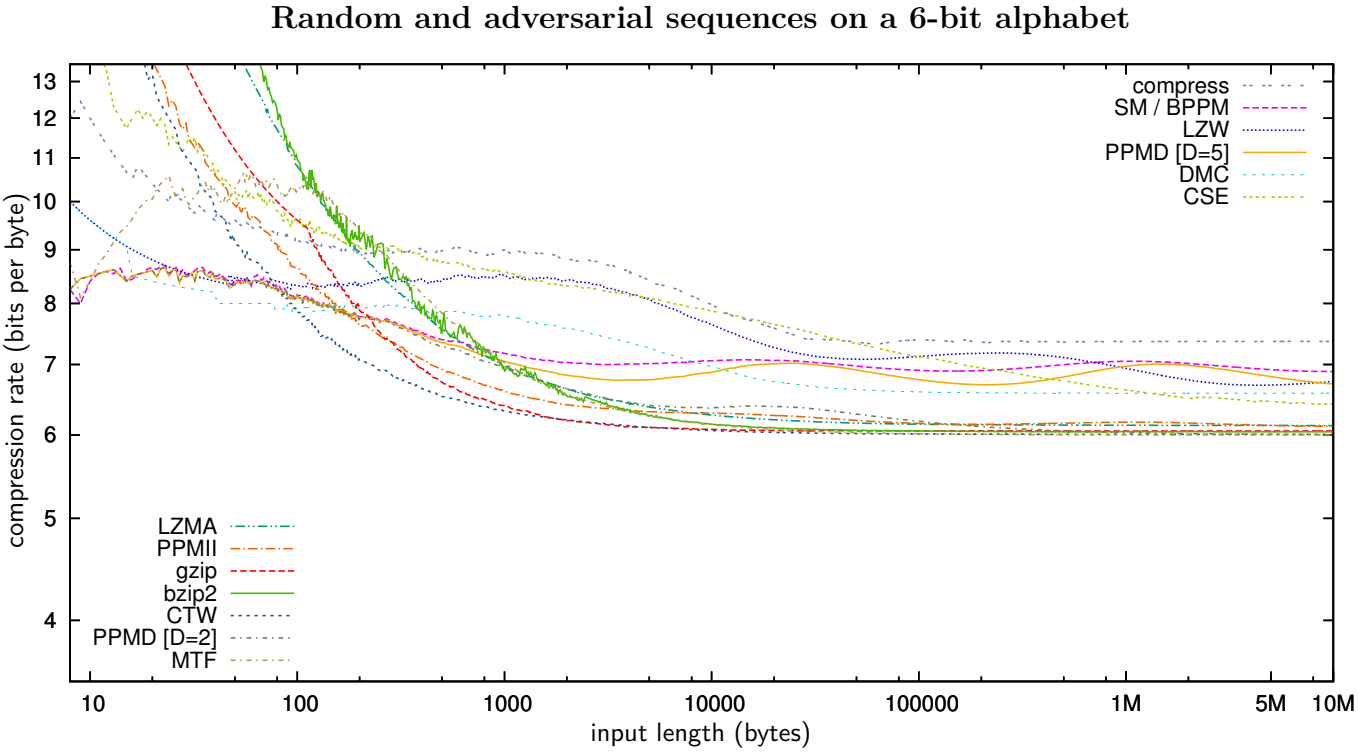
The compression rates were computed by compressing the first N symbols of the sequence, and were plotted on a log scale as a function of N (with N typically ranging from 10 to 10 000 000).

Key	Label		Description
.....	LZW	∗	LZW algorithm by Welch (1984), with arithmetic coding, uniform dictionary indices, and unbounded memory.
---·---·---	compress		Standard `compress` (LZW) by Thomas et al. (1985).
- - - - - -	gzip		Standard `gzip` (DEFLATE) by Gailly and Adler (1992).
———	bzip2		Standard `bzip2` (BWT) by Seward (2010).
-·-·-·-	LZMA		LZMA by Pavlov (2011), via `lzip` by Díaz Díaz (2013).
-·-·-·-	PPMII		PPMII by Shkarin (2001a).
- - - - -	SM / BPPM	∗	Sequence Memoizer / BPPM.
.....	PPMD	∗	Classic PPM, escape method D, various context depths.
···· ····	PPMD [D=1]	∗	Classic PPM, escape method D, with context depth $D=1$.
-·-·-·-	MTF	∗	Move-to-front encoding + adaptive index compression.
·-·-·-·	CSE		Prototype of "Compression by substring enumeration" by Dubé and Beaudoin (2010).
·····	DMC		Dynamic Markov compression by Cormack (1993).
·-·-·-·	CTW		Context tree weighting by Willems et al. (1993).

Table 8.5: Index of compression methods used in this chapter. Methods marked with ∗ are my own implementations. All other methods are implementations of the stated algorithm by the stated authors. Additional details and references can be found in Table A.1 (page 186).

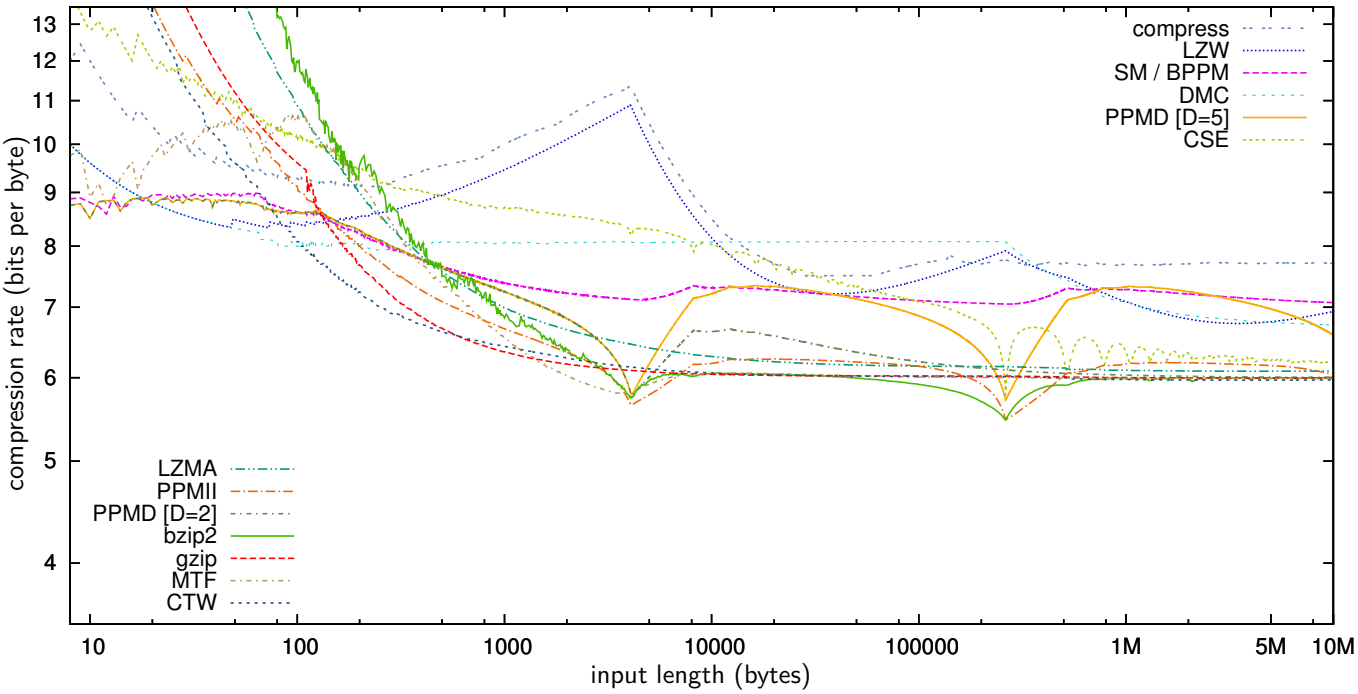
Seq	U	Symbol subset	Type	Figures
I	64	{␣, 0–9, A–Z, a–z, !}	uniformly random	8.4 (see also page 36)
II	64	{␣, 0–9, A–Z, a–z, !}	random adversarial	8.5
III	64	{␣, 0–9, A–Z, a–z, !}	alphabetic adversarial	8.6
IV	256	$\{00_{16}$–$FF_{16}\}$	alphabetic adversarial	8.7
V	256	$\{00_{16}$–$FF_{16}\}$	random adversarial	8.8 and 8.24
VI	256	$\{00_{16}$–$FF_{16}\}$	uniformly random	8.9 and 8.24
VII	32	{A–Z, @, [, \,], ^, _}	uniformly random	8.10
VIII	32	{A–Z, @, [, \,], ^, _}	alphabetic adversarial	8.11
IX	32	{A–Z, @, [, \,], ^, _}	random adversarial	8.12
X	16	{0–9, A–F}	alphabetic adversarial	8.13, 8.2 and 8.3
XI	16	{0–9, A–F}	random adversarial	8.14
XII	16	{0–9, A–F}	uniformly random	8.15 and 8.2
XIII	16	{0–9, A–F}	increasing integers	8.16
XIV	8	{0–7}	alphabetic adversarial	8.17
XV	8	{0–7}	uniformly random	8.18
XVI	4	{a, g, c, t}	alphabetic adversarial	8.19
XVII	4	{a, g, c, t}	uniformly random	8.20 (see also page 204)
XVIII	2	{0, 1}	alphabetic adversarial	8.21 (see also page 166)
XIX	2	{0, 1}	uniformly random	8.22
XX	16	{0–9, A–F}	alphabetic adversarial (LZW)	8.23 and 8.2
XXI	256	$\{00_{16}$–$FF_{16}\}$	random adversarial (LPAQ1)	8.24

Table 8.6: Index of synthetic sequences used in this chapter. U denotes the number of unique symbols in the sequence. With the exceptions of Seq. XX and Seq. XXI, all adversarial sequences were constructed with SM / BPPM as the victim.



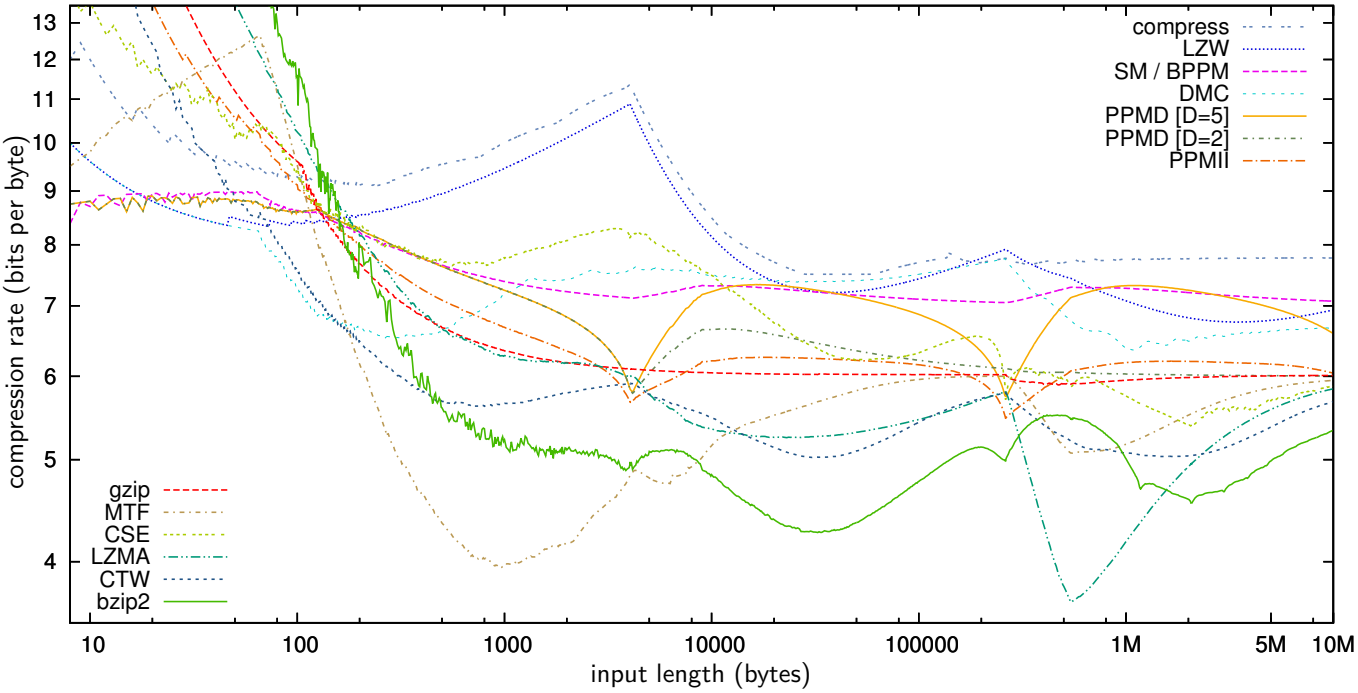
Start: `j2g2IxGif4uRMNGhSln!v8QRkwNoA8bKCFpMA9aGleZ4a4lo1JLBp␣Pmz4iXTsHDxbpBdNM7MQQeShTxT...`

Fig: 8.4 **Seq:** I **64 Symbols:** {␣, 0–9, A–Z, a–z, !}, **Type:** uniformly random.



Start: `jBYdU3Sr!VLT8xXJWcRzl2Ap4a9␣bnsODefGNFhgPo1ItZq60wKMCQ5vy7kmHuEiDhtdE2eiVZGFfxsLR...`

Fig: 8.5 **Seq:** II **64 Symbols:** {␣, 0–9, A–Z, a–z, !}, **Type:** random adversarial. **Victim:** SM / BPPM.

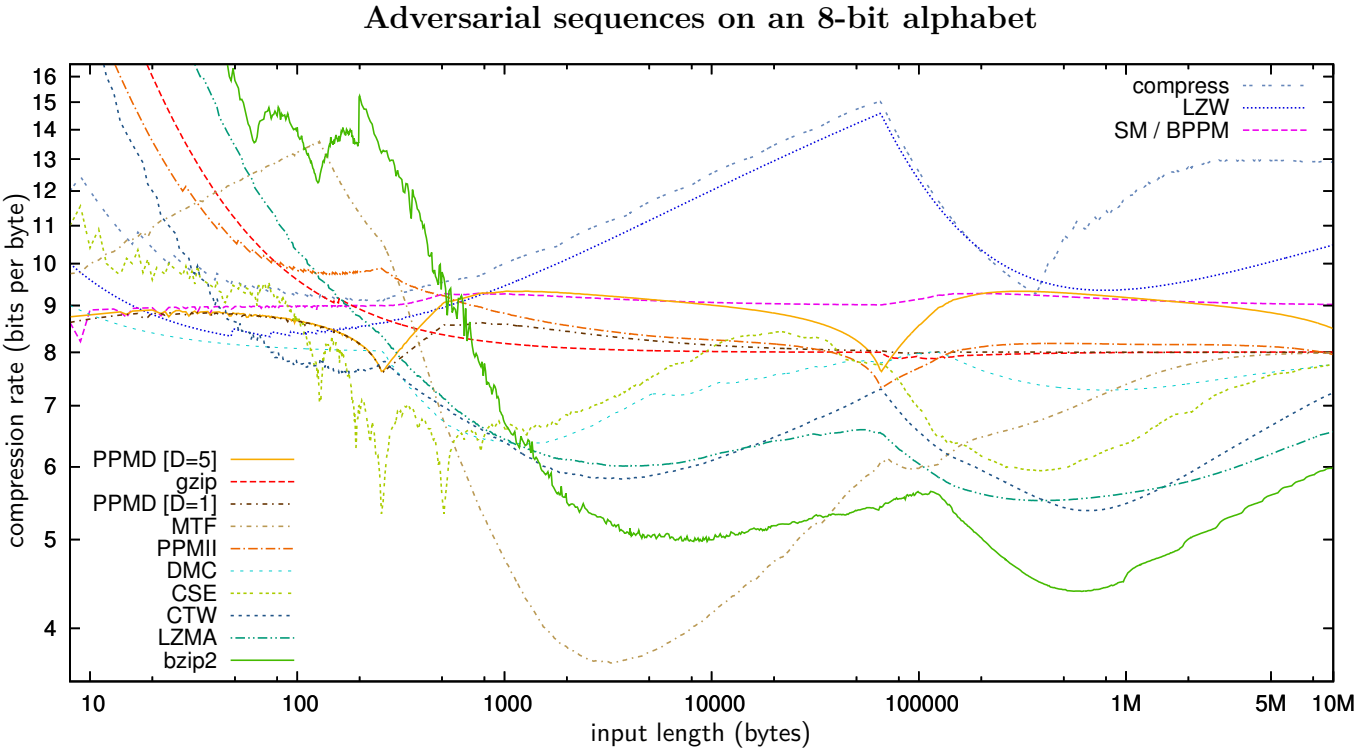


Start: ␣0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz!␣102435768A9BDCEG...
Fig: 8.6 **Seq:** III **64 Symbols:** {␣, 0–9, A–Z, a–z, !}, **Type:** alphabetic adversarial. **Victim:** SM / BPPM.

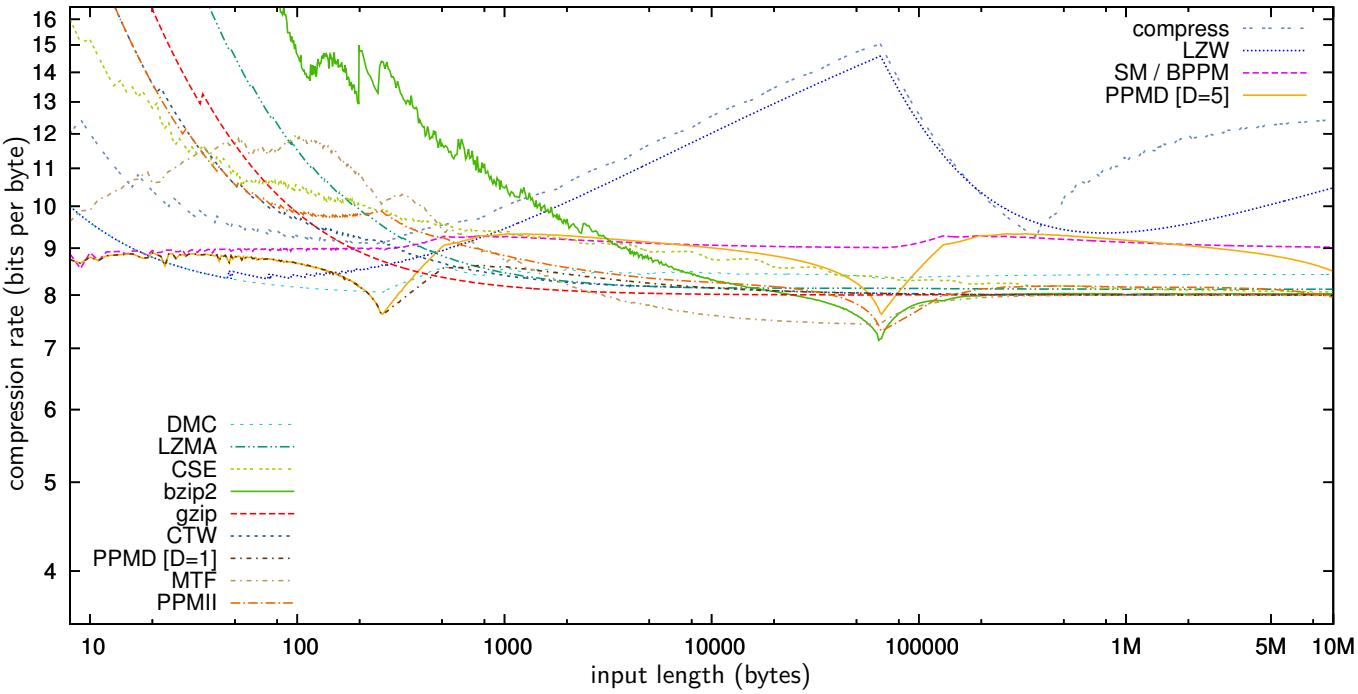
Figures 8.4, 8.5 and 8.6 show the compression effectiveness of selected compression algorithms on three special sequences. Seq. I is composed of random symbols that were drawn uniformly from a pre-selected subset of 64 (out of 256 possible) byte values. Seq. II is a random adversarial sequence for BPPM (resp. Sequence Memoizer with UKN approximation), where ties are resolved by chosing from the lowest probability symbols at random. Seq. III is also adversarial (for the same victim), but ties are resolved deterministically by choosing alphabetically, which makes the sequence potentially much more predictable for non-victim compressors.

On these adversarial sequences, and on Seq. III in particular, many compression algorithms exhibit fairly abrupt changes in compression effectiveness, often at particular positions of the sequence. In adversarial sequences for BPPM / Sequence Memoizer, these positions mark the locations where all possible n-grams (of a given n) have been generated.

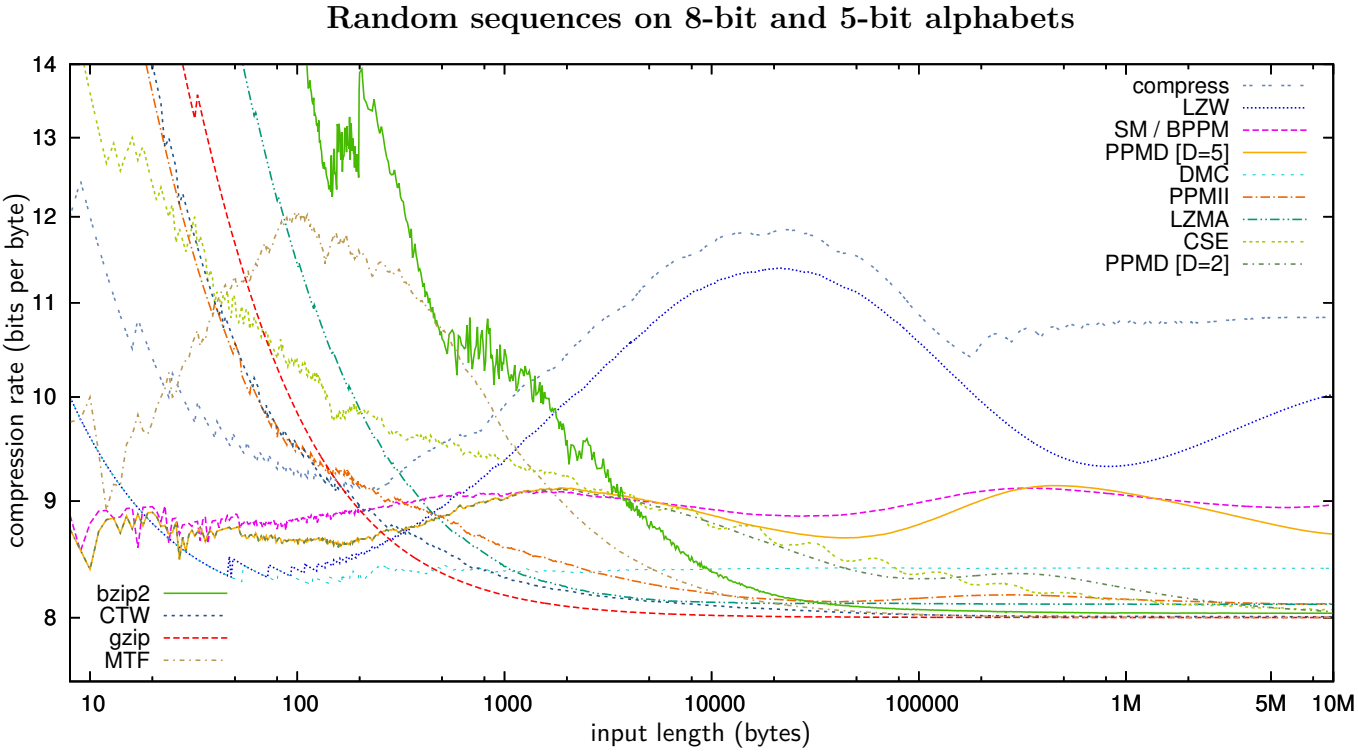
Compressors from the same family of algorithms tend to follow similar trajectories. For example, compress and LZW are independent implementations of the same algorithm. (LZW has a slight edge because it uses arithmetic coding for the dictionary indices. The point where the compression rate of compress levels out occurs when the algorithm reaches its built-in memory limit.)
For many compressors, the compression rate on sequences of uniformly distributed random symbols does not follow a monotone trajectory: there are hills and valleys that look a bit like smoothed out variants of those produced on the adversarial sequences.
The following pages showcase a series of similar graphs for random and adversarial sequences on differently sized subsets of the input alphabet, exhibiting some interesting results.



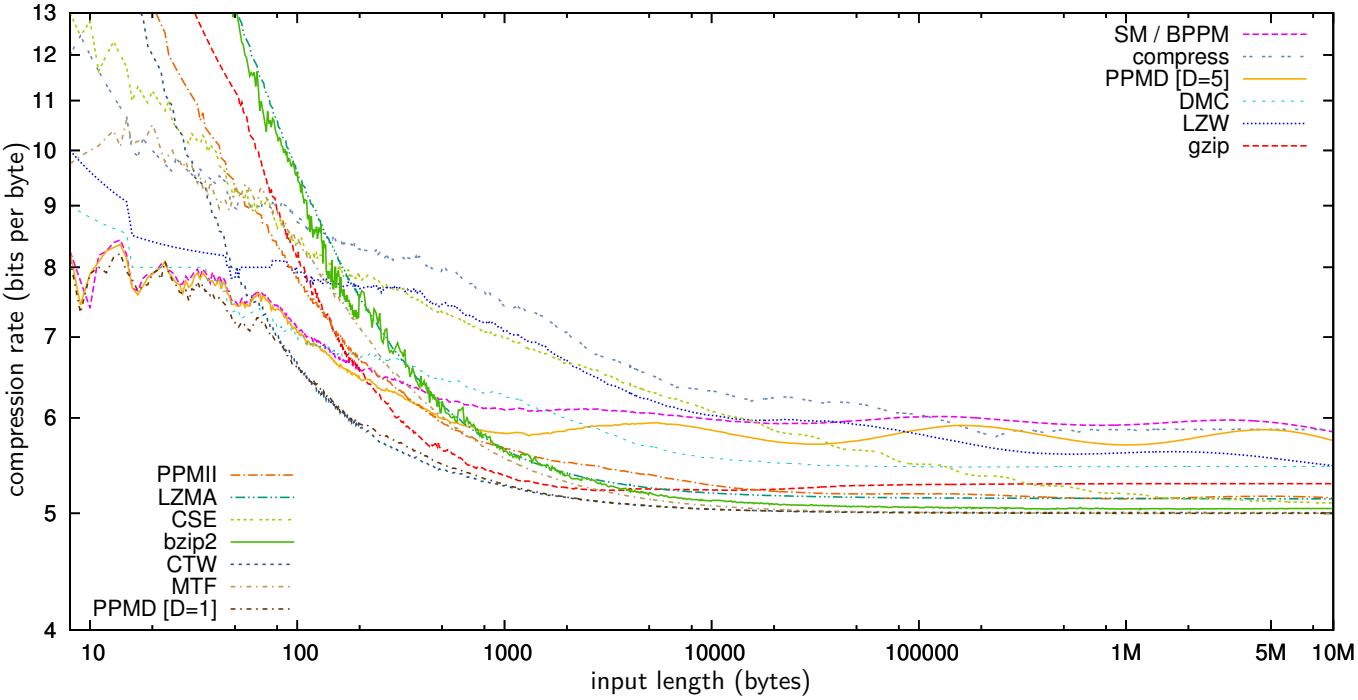
Start: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E . . .
Fig: 8.7 **Seq:** IV **256 Symbols:** {00_{16}–FF_{16}}, **Type:** alphabetic adversarial. **Victim:** SM / BPPM.



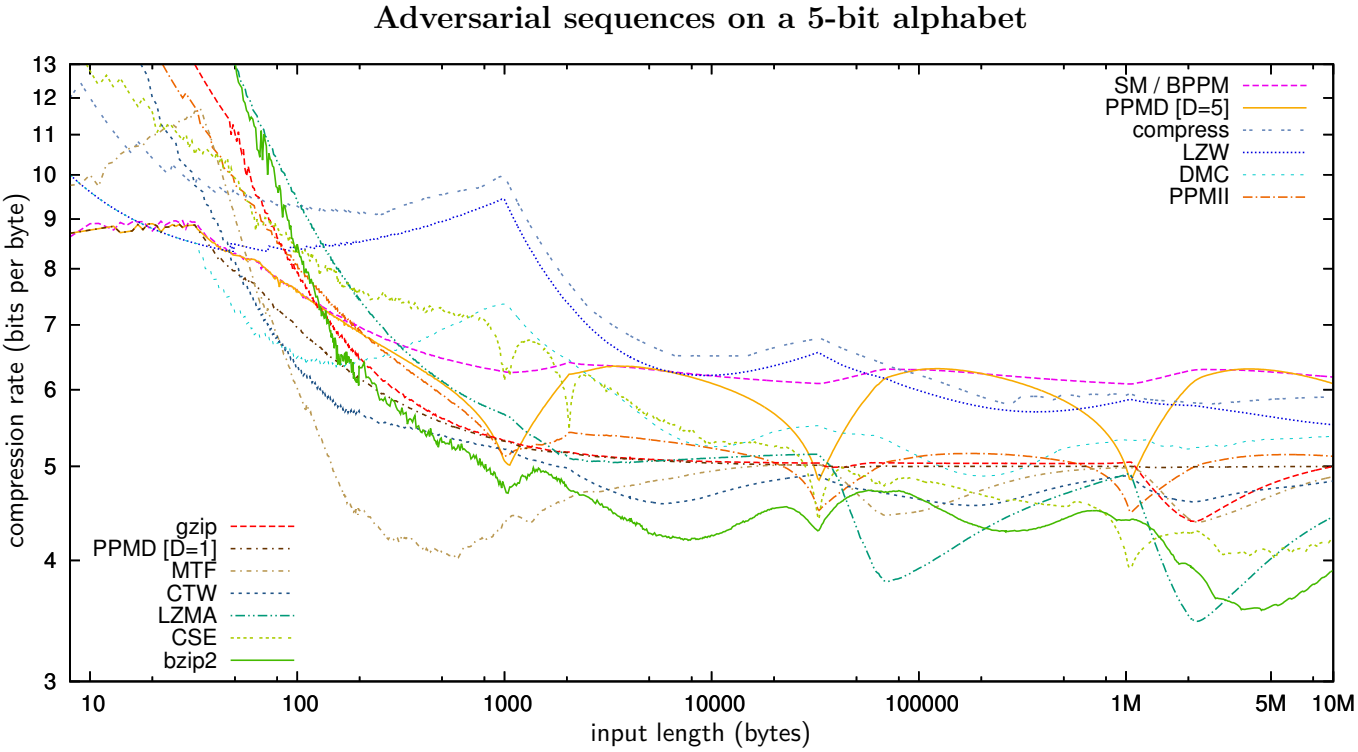
Start: BA C7 3C 95 36 A9 05 15 19 62 1D 90 5A 7D A7 01 E8 0B 65 7C AD 37 20 E7 C9 25 93 C8 4E 5C . . .
Fig: 8.8 **Seq:** V **256 Symbols:** {00_{16}–FF_{16}}, **Type:** random adversarial. **Victim:** SM / BPPM.



Start: 35 F7 E1 28 E4 4B 83 6D BE 9A BD DC 99 88 F8 20 11 54 0F E3 F1 6F 55 83 D6 52 DC 05 16 8C . . .
Fig: 8.9 **Seq:** VI **256 Symbols:** {00_{16}–FF_{16}}, **Type:** uniformly random.

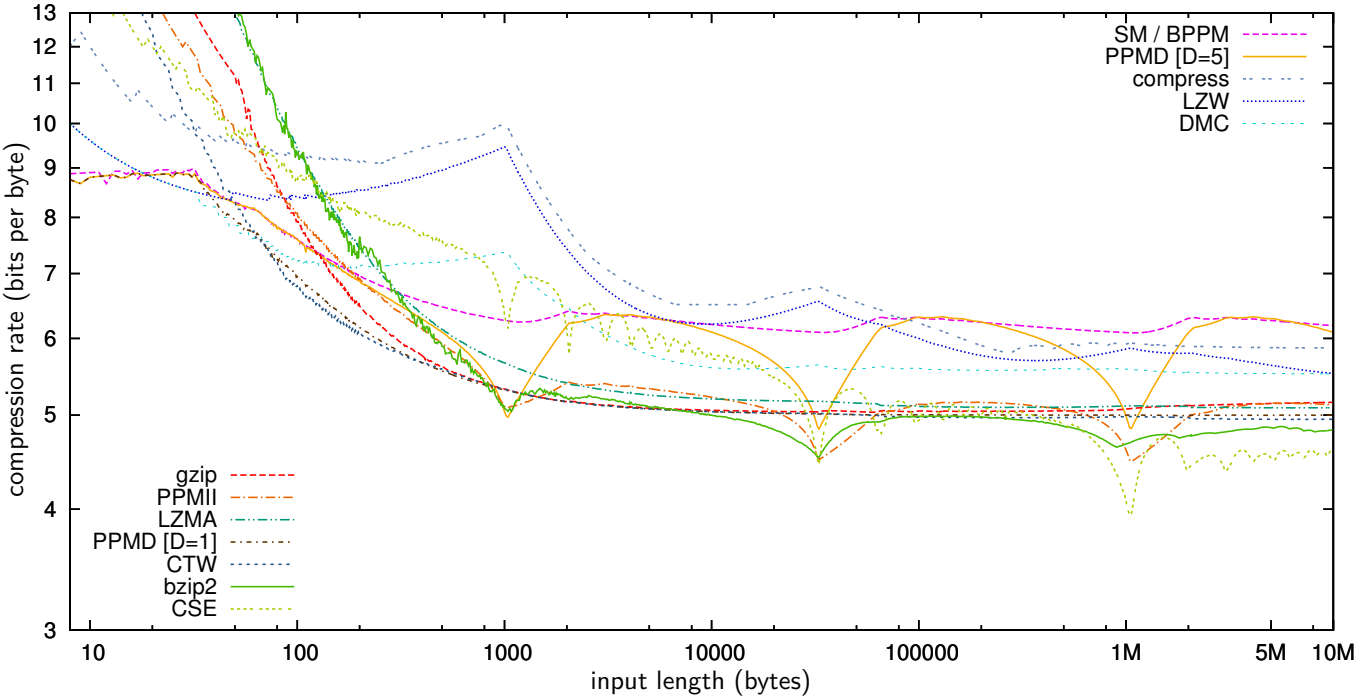


Start: `WAUAI^HVUB\NKLHVNXY_]DMNW]LYEDSJFHZKEERHXTRBRBXYAJKFZ@MX_BVQO[IG^SZFTLKDKMMTNVO^O...`
Fig: 8.10 **Seq:** VII **32 Symbols:** {`A`–`Z`, `@`, `[`, `\`, `]`, `^`, `_`}, **Type:** uniformly random.



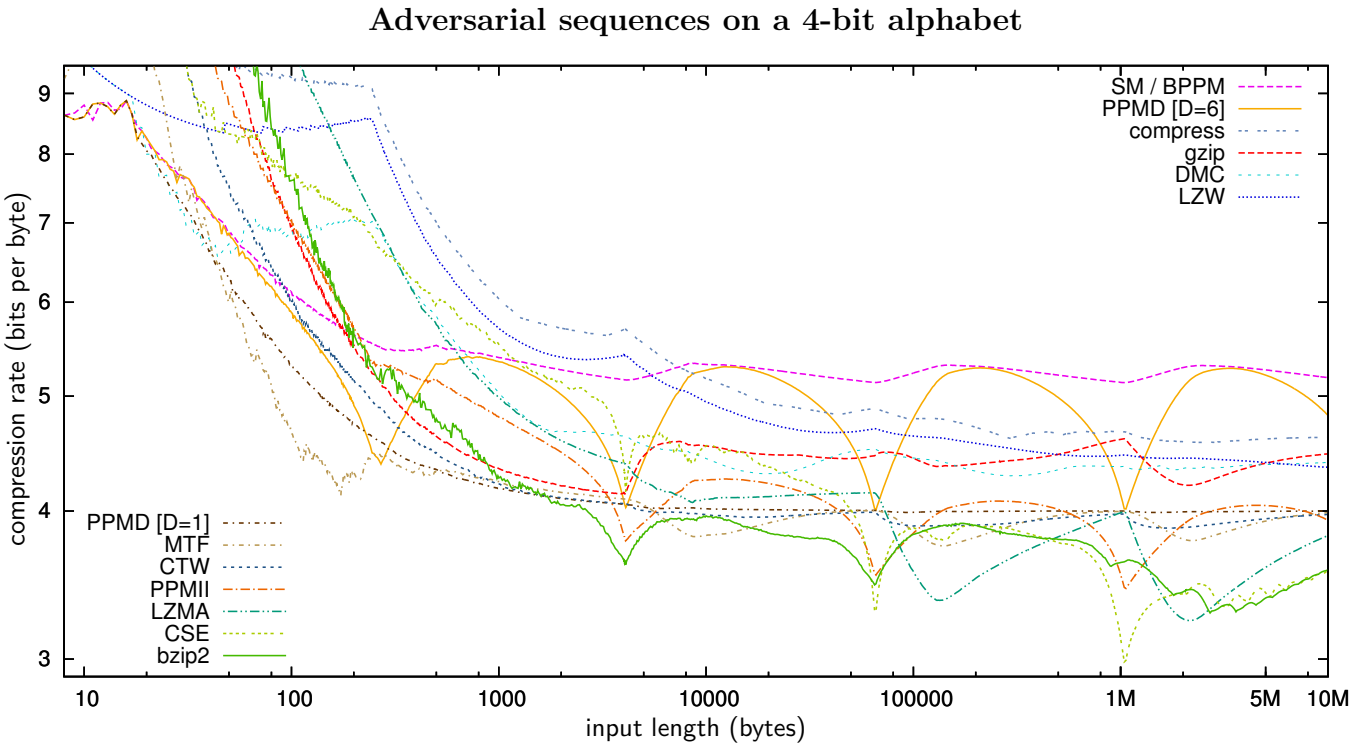
Start: `@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_@BACEDFHGIKJLNMOQPRTSUWVXZY[]\^@_ADBECFIGJHKMLONP...`

Fig: 8.11 **Seq:** VIII **32 Symbols:** {A–Z, @, [, \,], ^, _}, **Type:** alphabetic adversarial. **Victim:** SM / BPPM.



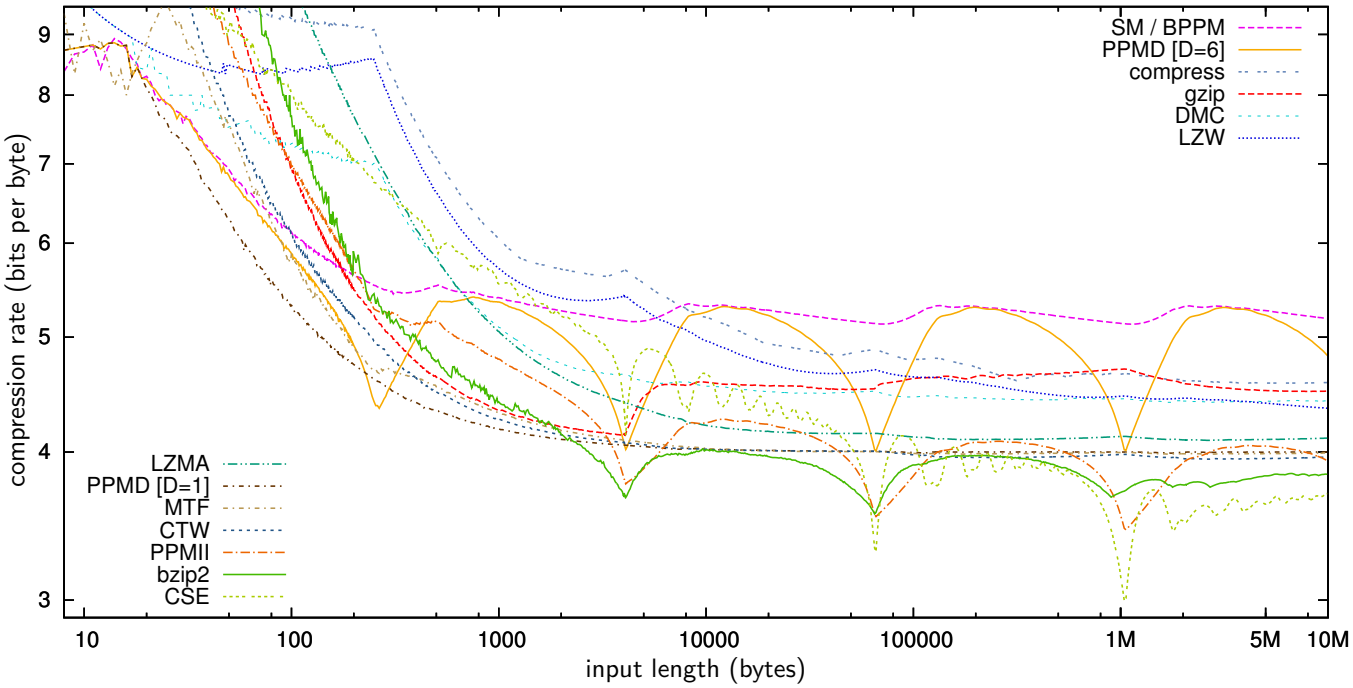
Start: `WJSP^RNXGT@O[ZK\IAM_UFQVLDYHBC]EJVRTPFYABUEXSWK[ING\Z]O_Q@DHML^CRWZ[OPS_VNQJTFXUGI...`

Fig: 8.12 **Seq:** IX **32 Symbols:** {A–Z, @, [, \,], ^, _}, **Type:** random adversarial. **Victim:** SM / BPPM.



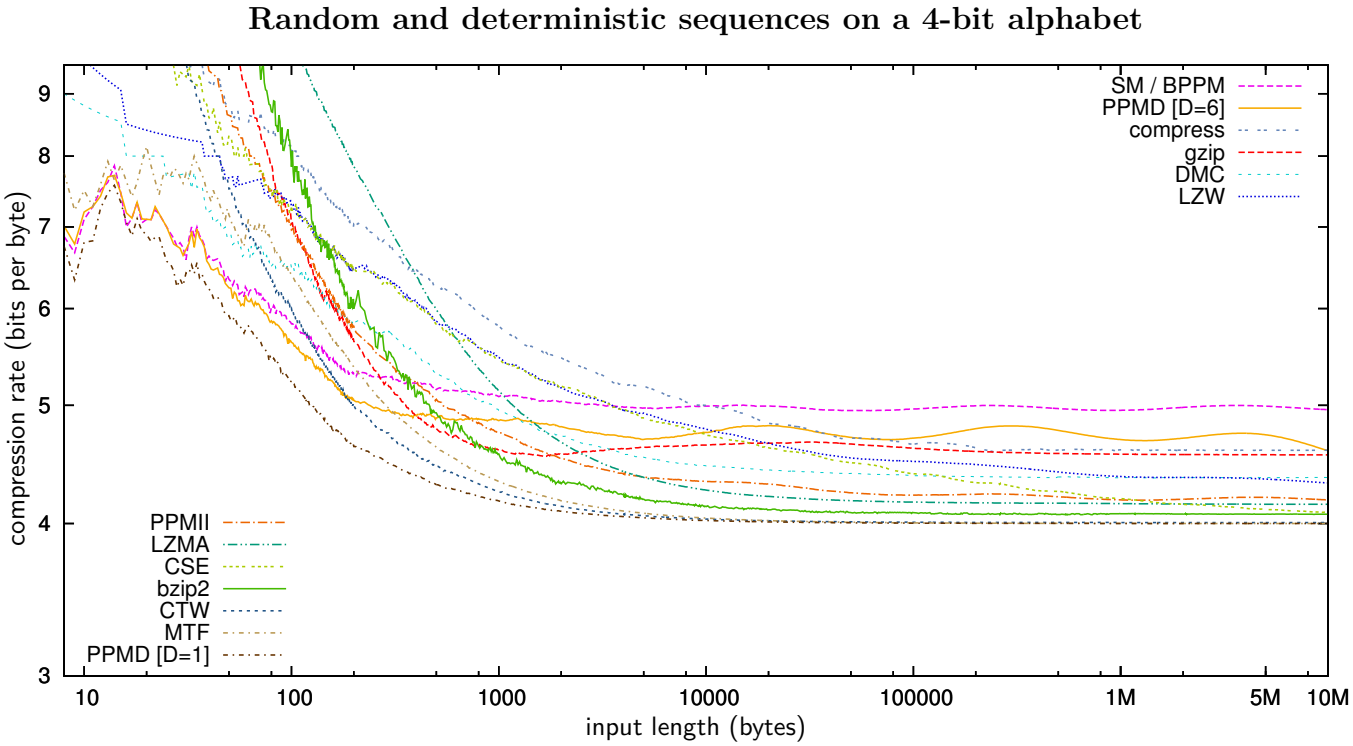
Start: `0123456789ABCDEF0213546879BACEDF1032475869CADBE0F251436A7B8C9D0E1F37264859EAFBDC0...`

Fig: 8.13 **Seq:** X **16 Symbols:** {0–9, A–F}, **Type:** alphabetic adversarial. **Victim:** SM / BPPM.



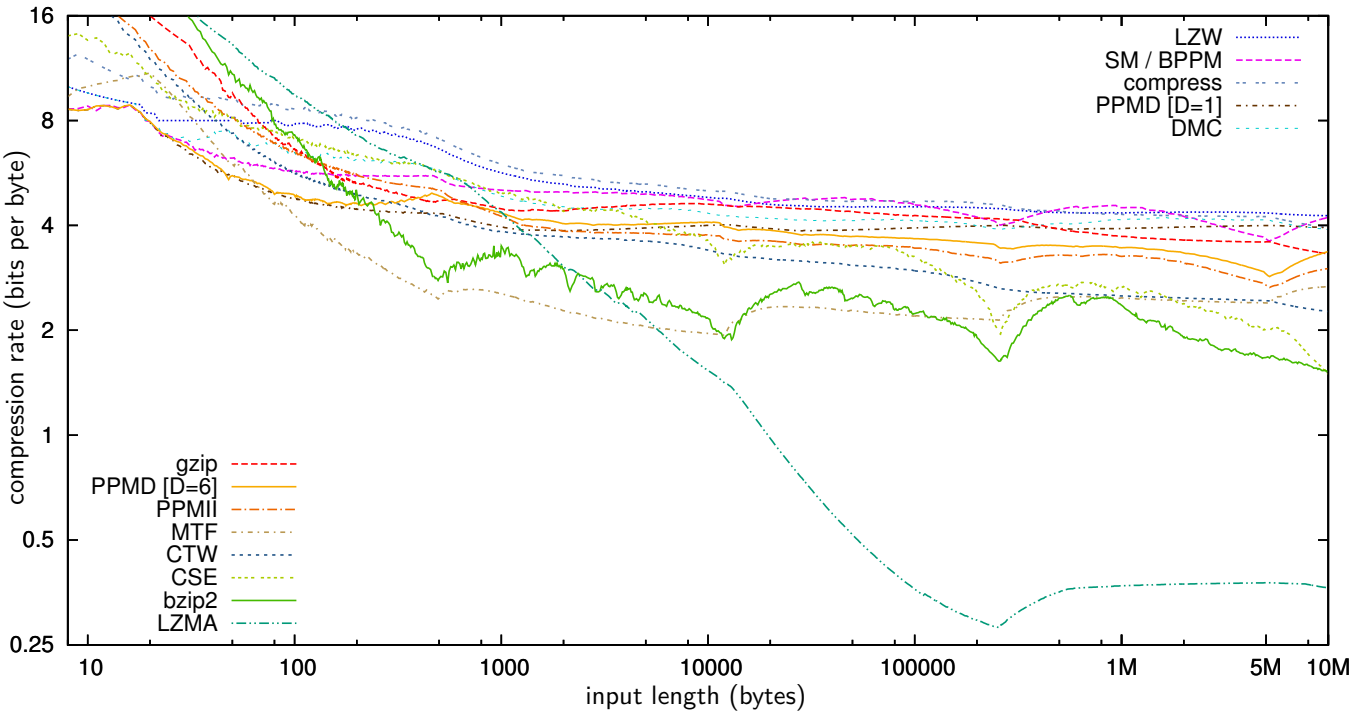
Start: `B37F8C62A54910DEB09AEF564183C72DAC26F9514ED087B134FCD59BA30E286715748FA6C029E3BD2...`

Fig: 8.14 **Seq:** XI **16 Symbols:** {0–9, A–F}, **Type:** random adversarial. **Victim:** SM / BPPM.



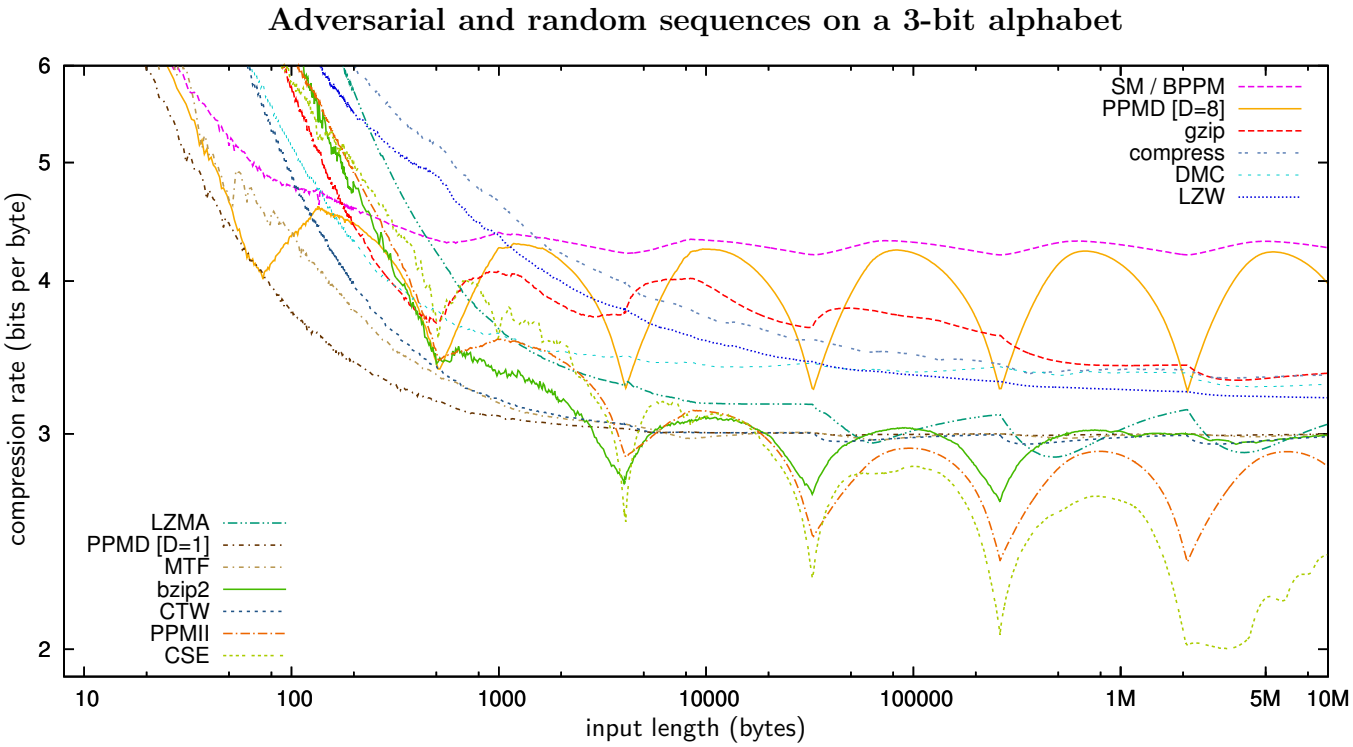
Start: `B0A04F4BA1E7564B7CCFE267BE6C229534D52294CA9191CC0553D06CF1B87D43F9D3A652566A7B7F7...`

Fig: 8.15 **Seq:** XII **16 Symbols:** {0–9, A–F}, **Type:** uniformly random.



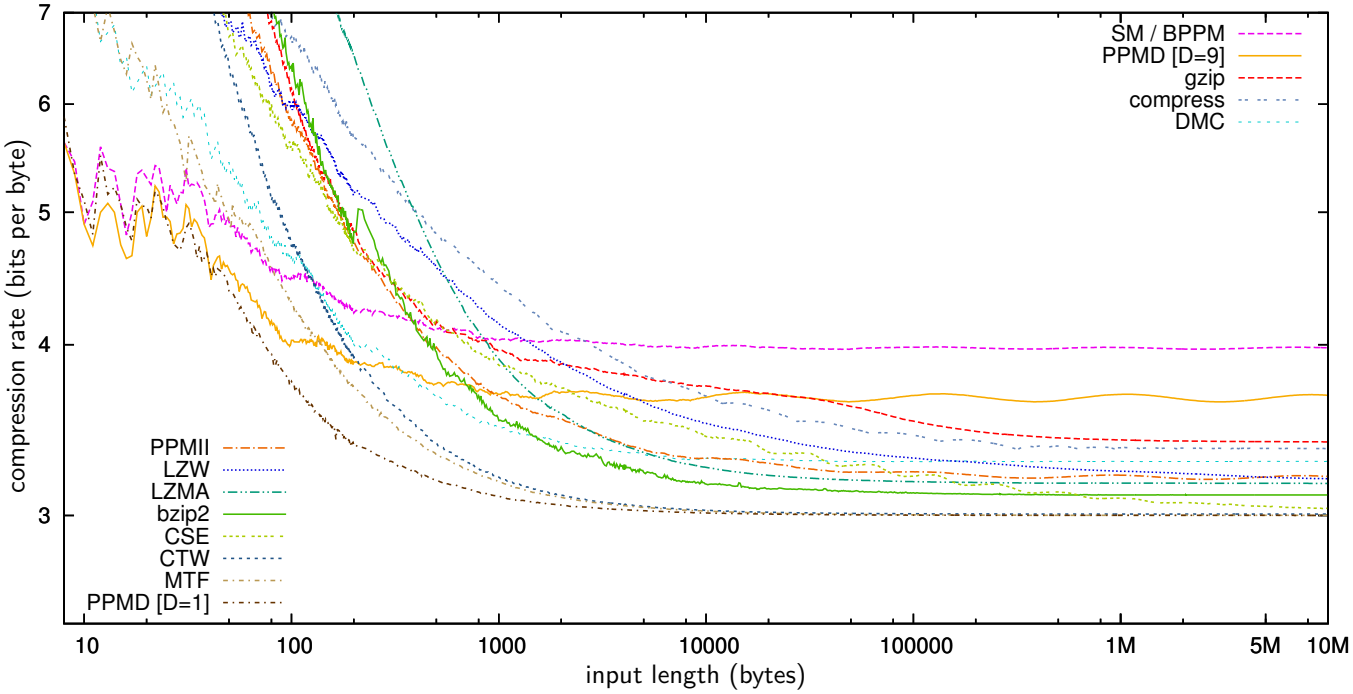
Start: `0123456789ABCDEF101112131415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F3...`

Fig: 8.16 **Seq:** XIII **16 Symbols:** {0–9, A–F}, **Type:** increasing hexadecimal integers.



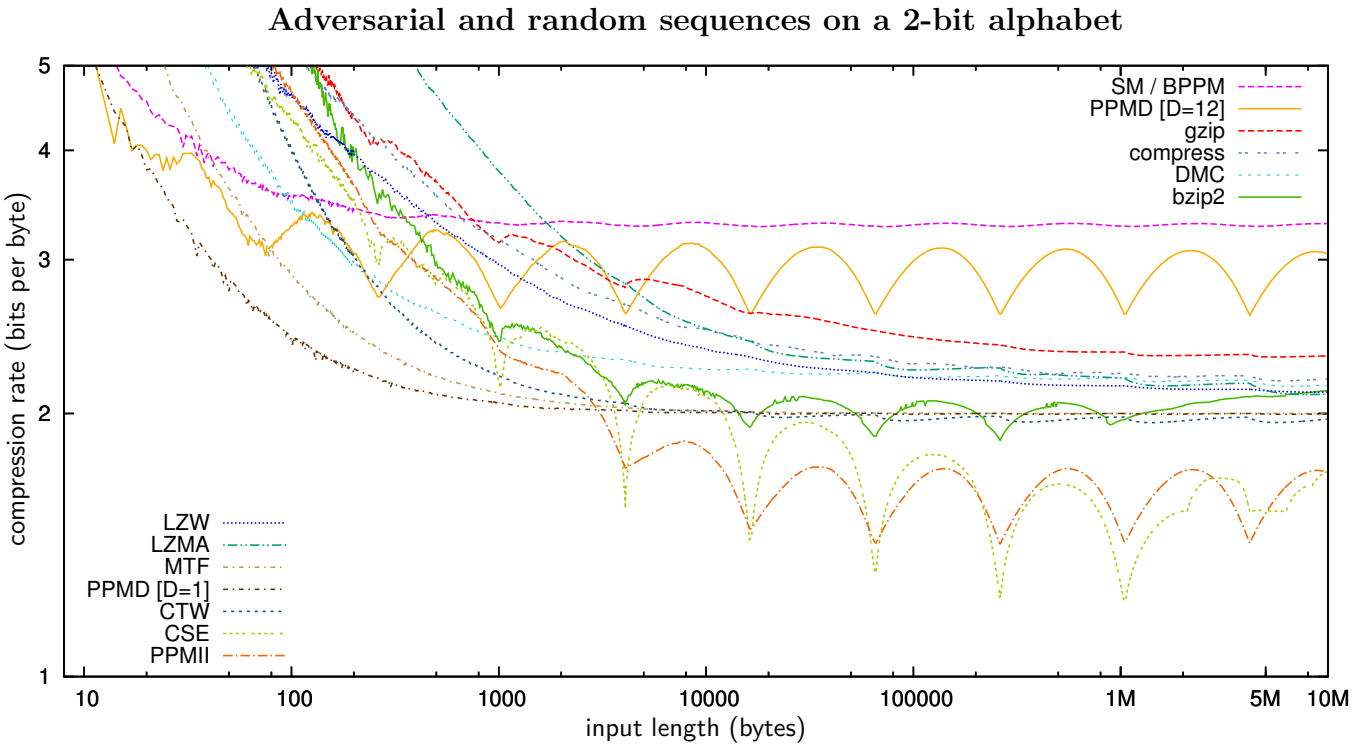
Start: 012345670213546071425361720374051624306573152647504100227632113344556677112231415 . . .

Fig: 8.17 **Seq:** XIV **8 Symbols:** {0–7}, **Type:** alphabetic adversarial. **Victim:** SM / BPPM.



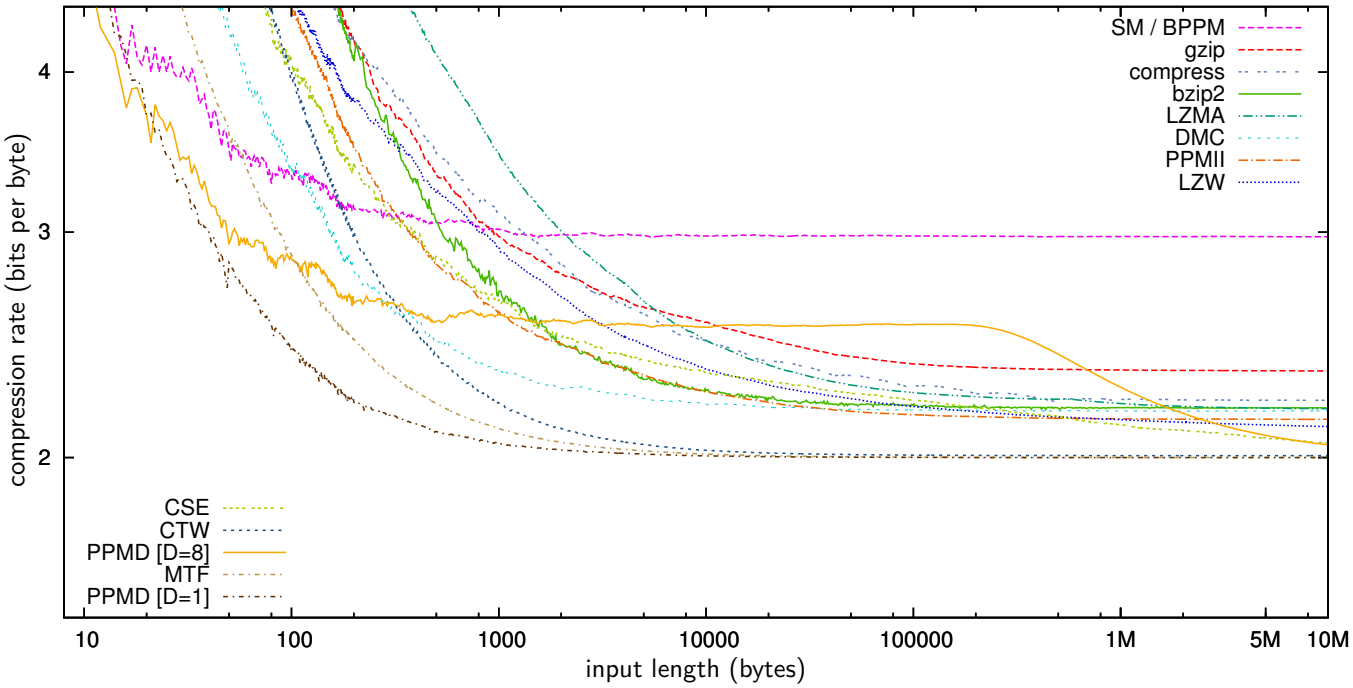
Start: 505027255073232536677133573611421262114265404066022160367054362174615321233535373 . . .

Fig: 8.18 **Seq:** XV **8 Symbols:** {0–7}, **Type:** uniformly random.



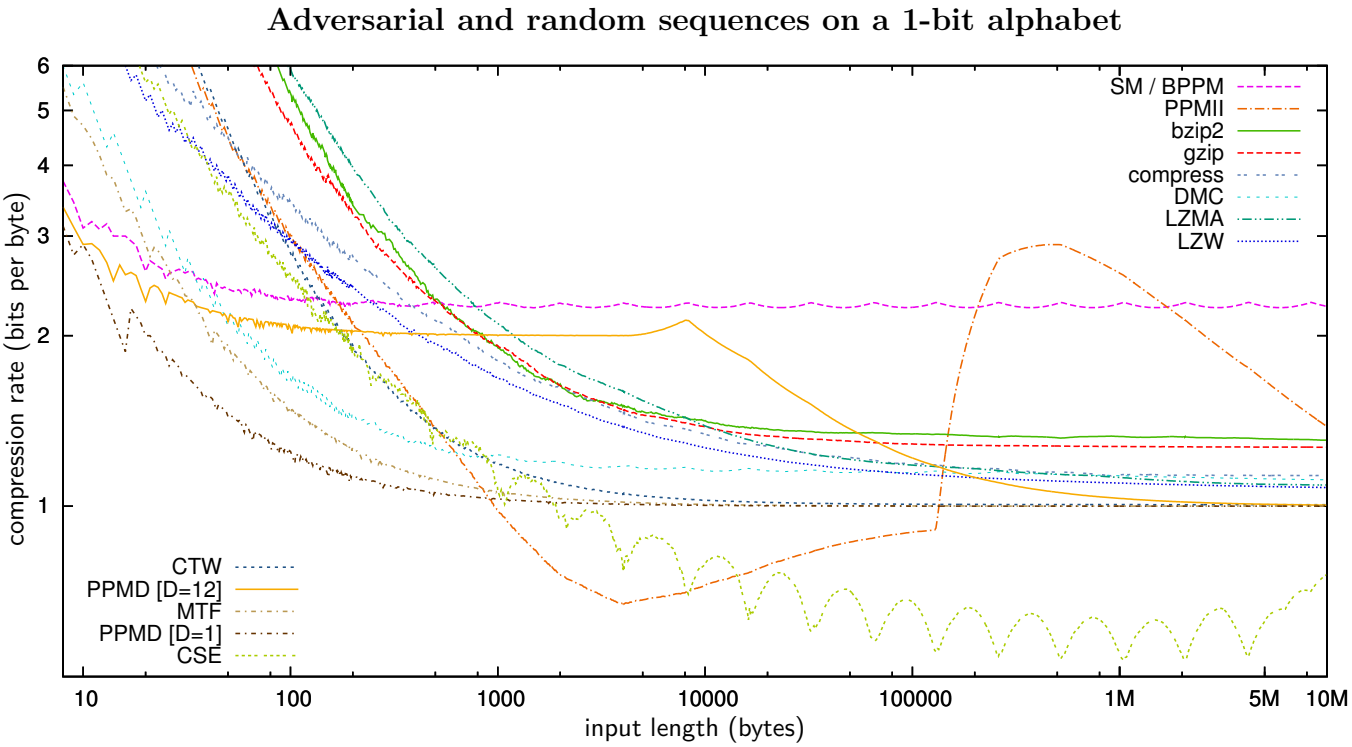
Start: `agctacgtgatcaaggccttggtcgaccattaactgcggagttccgcagaatatgtaggggctctttgcccacaaacggtg...`

Fig: 8.19 **Seq:** XVI **4 Symbols:** {a, g, c, t}, **Type:** alphabetic adversarial. **Victim:** SM / BPPM.



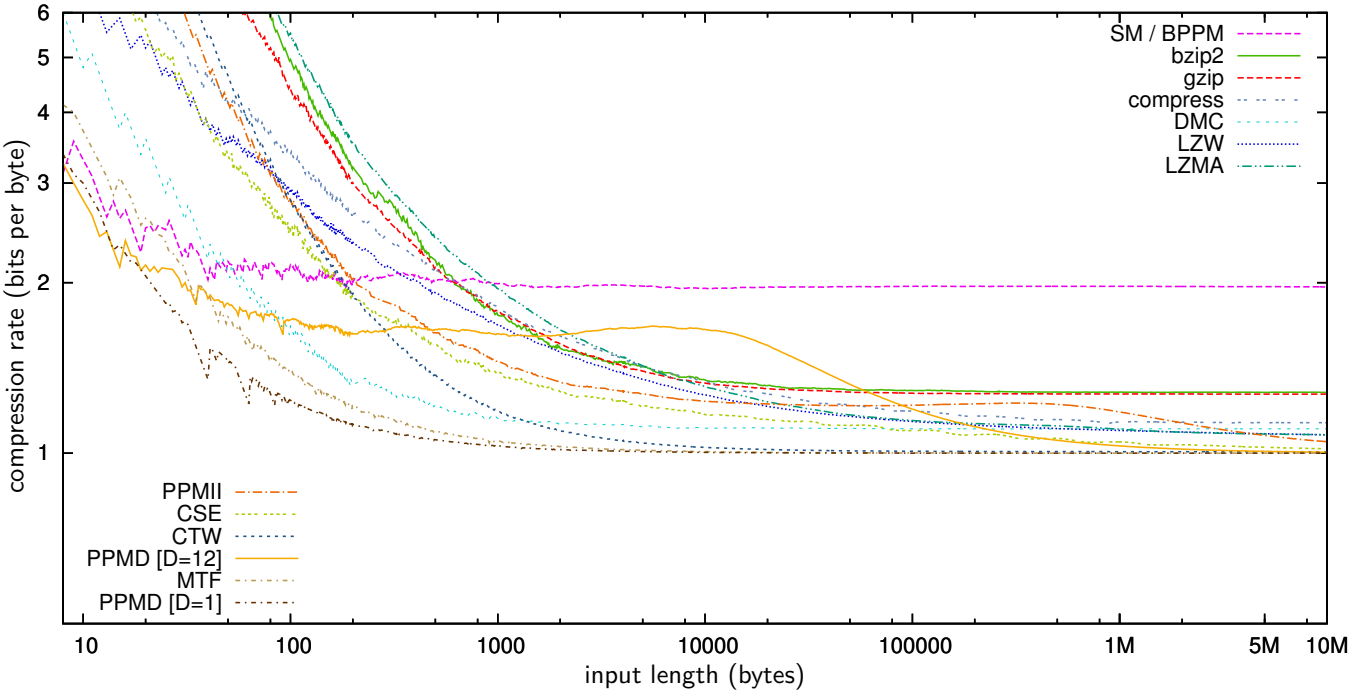
Start: `cacagtgccatggggcgttttaggctgtaacgagtgaacgtccacattaggatagttaccgtgatctacggagggcgcgtg...`

Fig: 8.20 **Seq:** XVII **4 Symbols:** {a, g, c, t}, **Type:** uniformly random.



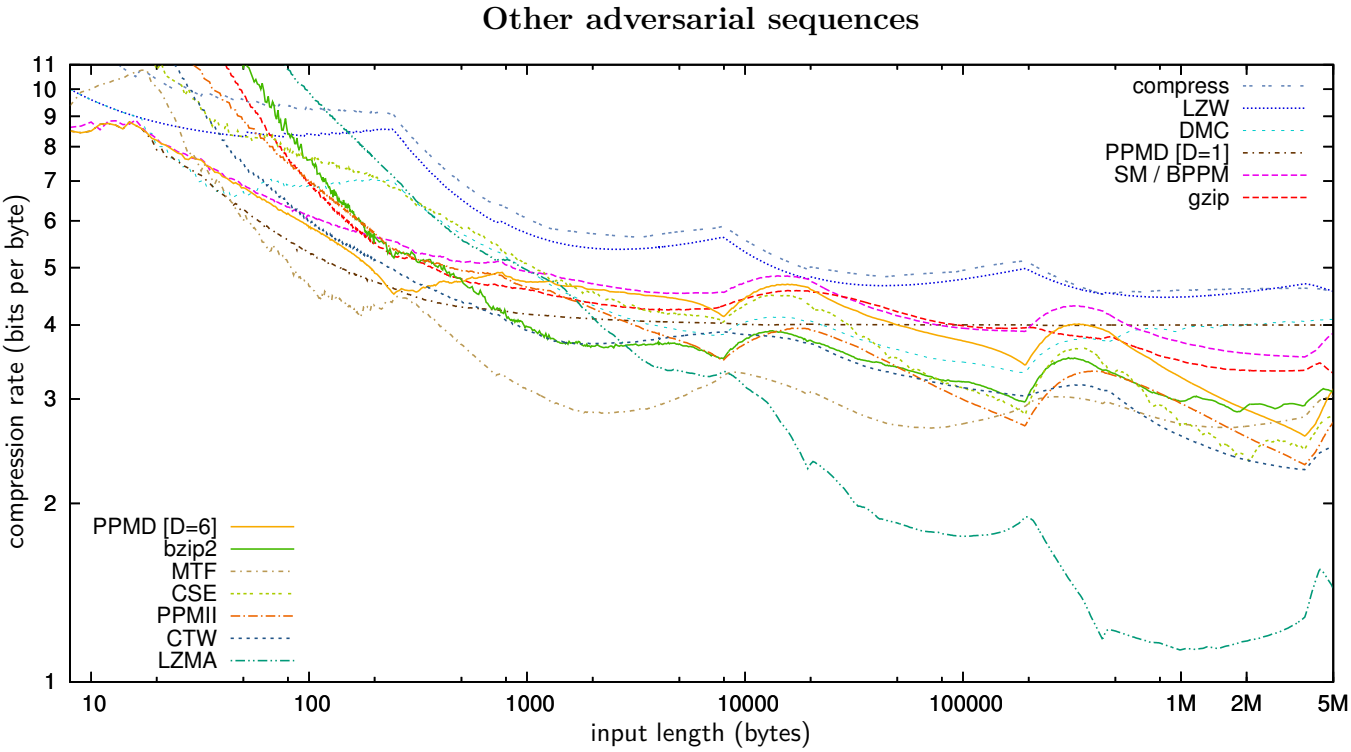
Start: 0100111011000010100011010111110010000001111010010110111000100110011111110111100000 . . .

Fig: 8.21 **Seq:** XVIII **2 Symbols:** {0, 1}, **Type:** alphabetic adversarial. **Victim:** SM / BPPM. *See page 166.*

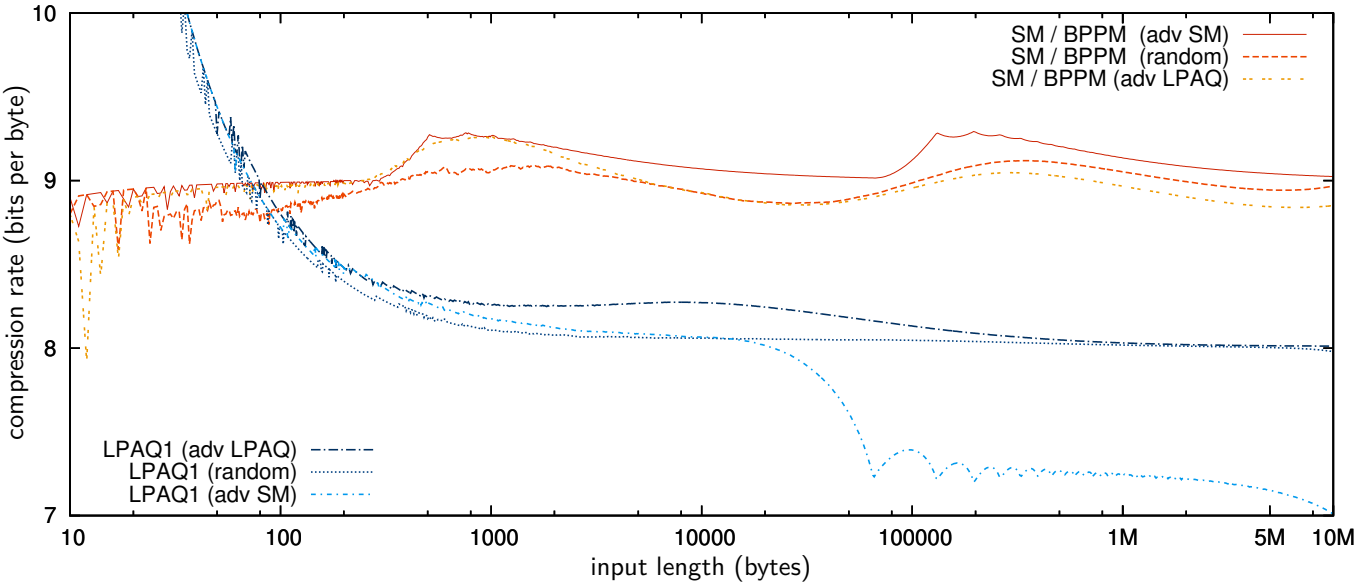


Start: 1010010110100001011110001101001000100010111010110000100110110100111010000000101010 . . .

Fig: 8.22 **Seq:** XIX **2 Symbols:** {0, 1}, **Type:** uniformly random.



Start: 0123456789ABCDEF0213546879BACEDF1032475869CADBE0F251436A7B8C9D0E1F37264859EAFBDC0 . . .
Fig: 8.23 **Seq:** XX **16 Symbols:** {0–9, A–F}, **Type:** alphabetic adversarial. **Victim:** LZW.



Start: DD B6 7A EA 3E 9E 1E F2 8A 56 CF FF A6 0E 6D 2F D5 4A E6 76 97 17 F8 27 C2 5D BA ED 36 . . .
Fig: 8.24 **Seq:** XXI **256 Symbols:** {00_{16}–FF_{16}}, **Type:** random adversarial. **Victim:** LPAQ1.

Figure 8.24 shows the compression effectiveness of two algorithms (BPPM and LPAQ1) on uniform random symbols (Seq. VI), on their own random adversarial sequence (Seq. V for BPPM, Seq. XXI for LPAQ1), and on each other's random adversarial sequence. For both algorithms, compressing their own adversarial sequence is harder than compressing randomness, which in turn is harder than compressing the other algorithm's adversarial sequence.

8.3.3 Discussion

Given access to a compression model's predictive symbol distributions, it is straightforward to construct sample sequences or adversarial input sequences for the model. However, many algorithms (such as LZ77, LZW, LZMA, CSE) do not explicitly compute or represent these distributions, making it difficult to construct adversarial sequences.

In principle, the predictive symbol distributions can be reverse-engineered from any given algorithm; I did this for LZW as a proof of concept, which provided a mechanism for creating Seq. XX. Obtaining LZW's predictive symbol distributions required significant structural alterations to the original algorithm (Alg. 2.3), and involved summing over partial string matches in LZW's dictionary entries.

It appears that some adversarial sequences have interesting properties that cause compression algorithms from different families (not just that of the victim) to exhibit strikingly different behaviours. Sequences that are adversarial to the Sequence Memoizer (and BPPM) seem to be particularly good at eliciting diverse reactions from different compression algorithms, and were therefore chosen as a primary focus in this chapter.

Not all compression methods have interesting adversarial sequences. For example, adversarial sequences for LPAQ1 (such as Seq. XXI) look very similar to sequences of uniformly distributed random symbols (to nearly all compressors except those from the PAQ family). Figure 8.24 shows the compression effectiveness of LPAQ1 and SM / BPPM on each other's and their own adversarial sequences, and on uniform random symbols.

Some observations are summarised below:

- For both random and most adversarial sequences, compressors from the same family have compression rates that tend to follow similar trajectories in the graph.
- Such graphs can reveal points at which a compressors's resource constraints are reached (e.g. maximum context depth, dictionary size, block size or memory usage).
- Several of the investigated adversarial sequences have locations at which the compression rate of nearly all algorithms undergoes abrupt changes – often in different directions. For the adversarial sequences of SM / BPPM, the nth such location corresponds to the point in the sequence where all n-grams (of the available symbols) have been generated.
- The alphabetic adversarial sequences seem to discriminate particularly well between different algorithms. This effect is particularly visible for Seq. III (Figure 8.6).
- Various compressors can be made to look good or bad just by truncating an adversarial sequence at a well-chosen point. Also, one could easily compute a sequence by picking

adversarially for one data model, and in favour of some other data model when breaking ties.

It is conceivable that adversarial sequences might be useful for detecting relationships between compression algorithms, or for analysing properties of a black-box compressor (an unknown closed-source implementation). Detectable properties may include algorithm family, parameter settings, and resource constraints.

Conclusions

This thesis reviewed various topics in lossless data compression, promoting an approach based on probabilistic modelling and arithmetic coding. Contributions include novel algorithms for losslessly compressing various kinds of input objects, and insights about existing compression methods for sequences.

For building any compression system, this thesis recommends designing a probabilistic model for the input data, defining an objective function that states the system's purpose, and building an algorithm that optimises the expected objective given all knowledge (the model and all available data). Separating the objective from the model is both elegant and useful, and provides a transparent way in which communication systems can be engineered and improved. In this thesis, the chosen objective function was to minimise the transmission cost of the output, and the algorithms used Bayesian inference and arithmetic coding.

Any probabilistic model specifies precisely which inputs it considers more probable than others, and is therefore necessarily biased towards certain kinds of input data – this bias is an unavoidable prerequisite for any form of compression. The aim behind *intelligent* models is to make that bias as flexible as possible, at a minimal cost: intelligent compressors can adapt to a wide variety of input data by learning from the data itself. Data compression has been proposed as a way of measuring artificial intelligence (Mahoney, 1999) that is more objective than a Turing test (Turing, 1950). The more intelligent the algorithm, the better it compresses (and vice versa).[1]

A natural question is then how well a "maximally intelligent" compression method would compress, and how such an algorithm might be constructed. The best possible compression achievable on a computer system is the length of the smallest computer program that generates the original data. This length is called the Kolmogorov complexity of the data (Kolmogorov, 1963, 1965, 1968; Solomonoff, 1964a). Unfortunately, Kolmogorov complexity (and hence also the smallest program) is uncomputable, making it impossible to find a theoretically minimal description of the input data with a computer.

Even if such a 'minimal generating program' could be inferred in practice, the decompressor could offer no guarantees about its operation, resource usage or termination, as decompression

[1] This relationship is a primary motivation behind the Hutter prize (Hutter, 2006).

involves executing an arbitrary computer program.[2]

As constructing a perfect general compressor is impossible, one may wonder how good our existing technology is, and if it can be improved. On English text, the best compressors currently yield an average of about 2 bits per symbol (see appendix A). Humans, as measured by e.g. Shannon (1950) or Cover and King (1978), predict English text with an average of 1.3 bits per symbol. Such results suggest that a gap in compression effectiveness remains between humans and machines, which innovations in probabilistic modelling and artificial intelligence should aim to close.

Of course, the utility of a practical data compression system involves not only compression effectiveness, but also compression efficiency (the speed of operation), resource usage (such as memory or energy), and how these properties scale as a function of the message size. When measured on these terms, machines easily win over humans: today's existing technology can communicate information much more effectively and efficiently than any human can. And hopefully, the reach and benefits of this technology will keep expanding.

I hope this thesis may contribute to the understanding and development of increasingly effective ways to store and transmit information.

[2]It is possible to approximate the uncomputable answer by placing careful restrictions on the program space that is being searched. Such an approach is explored by e.g. Hutter (2001, 2004), but remains computationally impractical for most real-world problems, including data compression.

Summary of contributions

Chapter 3 described an API for building compression algorithms based on arithmetic coding. The library follows an object-oriented design, and contains interfaces and algorithms for sampling, compression and inference; as well as implementations of many basic probability distributions, along with generic compression and sampling algorithms. All the work in this thesis was implemented with this API.

Chapter 4 described basic adaptive compression schemes for sequences, and contributed a novel compression method based on Pólya trees. The chapter also gave proof that online and header-payload compression, when implemented correctly, are mathematically equivalent.

Chapter 5 introduced generative models for data with various fundamental combinatorial invariances, including permutations, combinations, compositions, ordered partitions, sequences, and multisets. Compression schemes for these structures were derived by factorising the models into conditional univariate probability distributions, allowing the data to be arithmetically encoded in serialised form.

Chapter 6 contributed various insights into context-sensitive compression methods. For example, it showed the connections between the PPM algorithm of Cleary and Witten (1984a) and the Sequence Memoizer of Wood et al. (2009, 2011). It also gave a unified construction that includes several other related algorithms, showed the effects of their parameter settings and how to optimise them. It was shown that Deplump's compression effectiveness stems from the use of depth-dependent discount parameters rather than the use of unbounded depth.

Chapter 7 described methods for structurally compressing multisets of sequences based on a tree transform and binomial (or Beta-binomial) arithmetic codes. The compression achieved by this method exploits the disordered nature of the multisets, and therefore works even on multisets of incompressible sequences.

Chapter 8 showed how compression can be generalised to minimise the transmission cost of a message when the output symbols have different costs. The chapter also reviewed ways of using the predictive distributions of sequence compressors to sample or generate adversarial sequences. The properties of some of these sequences and the curious behaviours they induce on the compression effectiveness of existing compression models were documented and explored.

Appendix A contains an extensive comparison of compression algorithms on various standard corpora, especially PPM-like methods.

Future directions

The goal of chapter 5 was to give a comprehensive treatment of compression methods for basic combinatorial objects. There are several combinatorial structures that I wish I could have included; for example, a proper treatment of (unordered) integer and multiset partitions. An interesting generative model for integer partitions is given by the sampling formula by Ewens (1972), which can be derived also from the Chinese restaurant process of Aldous (1985). Unfortunately, I did not find a suitable factorisation into univariate variables for this distribution, and therefore was unable to build a suitable arithmetic coding scheme.

It would be nice to extend the work on structural compression to contain generative models for all combinatorial objects in the *Twelvefold Way* of Stanley (1986, section 1.9), also summarised in Knuth (2005b, section 7.2.1.4).

A promising research direction for the context-sensitive sequence compressors of chapter 6 that was not pursued is the use of fanout-dependent parameters, i.e. discount and strength parameters that are selected based on the number of unique symbols seen in the current node of the context tree; such an approach appears to be used by the PPMII compressor of e.g. Shkarin (2001a), and is motivated and described by Chen and Goodman (1998). Results for a prototype compressor built on this principle are included in appendix A.

There are several other probabilistic ways of modelling sequences that were not pursued in this thesis; examples include hidden Markov models and neural network models (Mnih and Hinton, 2009). Each of these could be used for building compressors, however I am not aware if these approaches have been tried. I attempted to include a fairly comprehensive comparison of the compression effectiveness of different compression algorithms; most of these can be found in appendix A.

Appendix A

Compression results

This chapter documents the compression effectiveness of many compression algorithms on files from several standard compression corpora. Most of these files were obtained from

`http://corpus.canterbury.ac.nz/` (Bell et al., 1998).

A list of all files, their sizes and SHA-1 hashes can be found in Tables A.2 and A.3.

The results found in this chapter were independently produced: nothing was copied from existing publications. For a result to qualify, each algorithm had to prove it could successfully decompress the compressed output it produced; luckily, all tested algorithms passed this test on all files. The compression rates were calculated based on the length of the compressed output (in bytes); that means that any overheads of the algorithm (file headers, rounding errors, padding to the next byte, etc.) are included in the numbers.

Hopefully, these numbers will help to compare these tested algorithms with each other, and also with new, yet undiscovered algorithms in the future.

Limitations. For a compressor to be of practical use, the resource usage and runtime characteristics of the algorithm are very important. These are not documented here, as many of the tested algorithms are research prototypes whose resource usage could easily be improved. Some of the modifications that are necessary for making an algorithm practical (such as imposing memory limits, or guarding against overflow errors) can adversely impact the compression effectiveness; these are trade-offs that any sensible implementation will necessarily have to make.

The compression efficiency (speed of operation) varies significantly among the compressors tested here. Some of the algorithms with the best compression effectiveness (notably those from the PAQ family) are also very slow, and can take significantly longer to run than other algorithms.

Table A.1: Index of compression algorithms used in this thesis. Each row identifies one concrete compression method.

Name	Type	Description
HC(en)	HC	A fixed Huffman code built for letter occurrences in English text.
CRP	CRP	A single, two-parameter Chinese Restaurant Process as described in section 4.2.3 (with $\alpha=0$ and $\beta=\frac{1}{2}$), followed by arithmetic coding.
Polya	Polya	The Pólya Tree symbol compressor, as described in section 4.3.1.
MTF	MTF	Basic move-to-front encoding, followed by adaptive compression of the indices (using arithmetic coding).
LZW	LZW	Unbounded memory LZW with uniform distribution over dictionary indices, and arithmetic coding.
compress	LZW	Unix `compress` by Thomas et al. (1985). Version 4.0, compiled for Linux Source: `compress.c` **SHA-1:** `290066c3a327521841fb25b3172c5a6dd9c22c08`
pkzip	LZ77	DEFLATE implementation by Katz and Burg (1993), for DOS. Exec: `pkzip.exe` **SHA-1:** `a3e401daa2bba1999d4a7c94f5d8099d2f053872`
gzip	LZ77	DEFLATE implementation by Gailly and Adler (1992), version 1.4 [Linux], open source. `gzip -9`.
7zip	LZMA	7-Zip compressor by Pavlov (2009), version 9.04 beta [Linux].
lzip	LZMA	Command line frontend by Díaz Díaz (2013) to the LZMA algorithm by Pavlov (2011). `lzip`.
PPMA d	PPM	Standard PPM with escape method A and max. context depth $D=d$.
PPMD d	PPM	Standard PPM with escape method D and max. context depth $D=d$.
PPME d	PPM	Standard PPM with escape method E and max. context depth $D=d$.
BPPM	PPM	Finite depth PPM with *blending*, as defined in section 6.5.
BM-J d	PPM	BPPM with global parameters $\alpha=0.5$ and $\beta=0.75$, see page 140.
BM-K d	PPM	BPPM with global parameters $\alpha=0.5$ and $\beta=0.85$, see page 140.
DDA d	PPM	BPPM with 7 depth-dependent parameters: α_1 ... α_7 = (14.67, 0.83, 0.44, -0.11, 0.21, -0.0038, 0.76) β_1 ... β_7 = (0.006, 0.56, 0.74, 0.79, 0.87, 0.89, 0.94).
FDA d	PPM	BPPM with 4 fanout-dependent parameters: α_1 ... α_4 = (0.5, 1, 2, 4) β_1 ... β_4 = (0.739, 0.836, 0.835, 0.831).

Name	Type	Description
FDB d	PPM	BPPM with 6 fanout-dependent parameters: α_1 ... α_6 = (0.5, 1, 2, 4, 5, 6) β_1 ... β_6 = (0.769, 0.832, 0.843, 0.854, 0.804, 0.775, 0.818).
ppmz2	PPM	PPMZ 2 by Bloom (1998), official implementation. [win32] Exec: `ppmz2.exe` **SHA-1:** `82dd192344a3cc8baa6be2abaec4be504c751777`
PPMII	PPM	Reference implementation of PPMII, by Shkarin (2006). [win32] Exec: `ppmd.exe` **SHA-1:** `f54e20319ba9a5fc2ff052721df264e419a211cb`
SM-0	SM	Deplump (my own implementation), with parameters $\alpha = 0$ and $\beta = \frac{1}{2}$.
SM-JG	SM	Deplump (my own implementation), with parameters from Gasthaus (2010): UKN, $\alpha = 0$, β_1 ... β_5 = (0.62, 0.69, 0.74, 0.80, 0.95).
bzip2	BWT	Open source compressor by Seward (2010), version 1.0.6 [Linux]. `bzip2 -9`.
CSE	CSE	Compression by substring enumeration (Dubé and Beaudoin, 2010), prototype.CSE Exec: `butterfly` [Linux] **SHA-1:** `46f9f9ffecca28585e43f815a895a418add06568`
CTW	CTW	Context tree weighting by Willems et al. (1993), official implementation. Exec: `ctw.exe` [win32] **SHA-1:** `0e8ec17da921b5c841e56ea383c2fe7978991283`
DMC	DMC	Dynamic Markov compression by Cormack (1993), official implementation. Source: `dmc.c` [Linux] **SHA-1:** `d2d574c6cecc39c7d2266b7b0616cb7df5b89592`
lpaq1	PAQ	Lightweight version of the PAQ compressor, by Mahoney (2007a). [Linux] Source: `lpaq1.cpp` **SHA-1:** `1f056441997daee6f56500f7ab1cd26aa9373c4e`
paq8l	PAQ	PAQ8L compressor by Mahoney (2007b). [Linux] Source: `paq8l.cpp` **SHA-1:** `9b96c5b4beb787905689e77b3831ba0d891158c1`

The family of dictionary compressors is described in section 2.3, with added detail for LZW in section 2.3.1. BWT is covered in 2.4.2. The standard PPM variants are described in section 6.3, and PPM escape methods are summarised in Table 6.2. BPPM and SM are examined in sections 6.5 and 6.6. CSE, DMC, CTW and PAQ are not discussed in detail in this thesis, but results and references to literature are provided. The keyword index contains relevant pointers for each algorithm.

Table A.2: Index of corpora for benchmarking file compression. The corpus of choice in this thesis is the Canterbury corpus, but supplementary results for the Calgary corpus are included to allow comparisons with results from older publications.

The Canterbury Corpus (Arnold and Bell, 1997)

File	Size	SHA-1 sum	Description
alice29.txt	152089	37a087d23c8709e97aa45ece662faf3d07006a58	English text
asyoulik.txt	125179	fb7db2d0c1ba0a1be26fe1892a7f83bf01153770	Shakespeare
cp.html	24603	fc4c10407efe47f40eee55eba9bddffbe5948cf4	HTML source
fields.c	11150	31999f829d313a6c10314deba2854b23481ab346	C source
grammar.lsp	3721	12bf64bf1d4c1f1119bea24e7bebd3167389220d	LISP source
kennedy.xls	1029744	bb3c73adde28228f9a311ddfee8f76aeccf83c4b	Excel spreadsheet
lcet10.txt	426754	77331951a4e24cd6d6315aa41b2cbdda882f685d	Technical writing
plrabn12.txt	481861	4575958b534bbe6e9d461b0b390300f54a5210cd	Poetry
ptt5	513216	96f7ab3d975ea4d823cf30be7dad5827f45858e9	CCITT test set (fax)
sum	38240	076fb264487f2d6868e67919e1949dcb4560e423	SPARC executable
xargs.1	4227	777250a5ccf4fd95b48c1c9248ab82c2e0221913	GNU manual page

The Calgary Corpus (Bell et al., 1990)

File	Size	SHA-1 sum	
bib	111261	3f86203a59b9d823c784f0414dd1920bcb62d067	Bibliography
book1	768771	673c583d45544003eb0edd57f32a683b3c414a18	Fiction book
book2	610856	b855cfafe7374942a0ae54c3bd90f0bce7b73fab	Non-fiction book (troff)
geo	102400	5cf652cfcc8e556ffb5e118fc29bcffef0aa71ab	Geophysical data
news	377109	afd9f190c621f45216a321485a543c00786bc76b	USENET batch file
obj1	21504	d155a7f8c68d24e9914b3274d5b5a6aa720e8d58	VAX object code
obj2	246814	e02c588e271f242fd00ecc68a931d9c5485323a0	Apple Mac object code
paper1	53161	aef6dac8838b1e9b35a46a6c1ccf1876a63486b4	Technical paper
paper2	82199	93d9bf0d3b4eae5198cf589336b30af3d6607feb	Technical paper
pic	513216	96f7ab3d975ea4d823cf30be7dad5827f45858e9	CCITT test set (fax)
progc	39611	66fa53f757f6474ad92b2ae52ef07981839dd14d	C source
progl	71646	7f9723167476639998ece850b9fbe1e5587aed1c	LISP source
progp	49379	22c59a4046cc52510a736582eae4bcdca4713411	PASCAL source
trans	93695	34322336c2c5b210fefc5b85517c65de1d184da5	TTY transcript

Other Files (various sources)

File	Size	SHA-1 sum	
world192.txt	2473400	fe5b97b714b2abe91a5e64f4e9b4589f61a6a45e	The CIA world fact book
shakespeare.txt	5283795	8ef376bc97ea79b1fa5dd1aaa10c4b5d82a1be95	All of Shakespeare's works
kokoro.txt	484562	e906613d135c80f4ee7e9b053d7b3f79bfb56552	Japanese text

Table A.3: Index of corpora for benchmarking compression of DNA sequences, mathematical sequences and random sequences.

A DNA corpus (Grumbach and Tahi, 1993, 1994)

File	Size	SHA-1 sum	
chmpxx.chr	121024	4b310bb85e086d1098e29f958aa37a8e4c3eeb32	Marchantia polymorphia chloroplast
chntxx.chr	155844	8148ecf34d93db8a3354bc2f1573600a399ac2e9	Tobacco chloroplast
hehcmv.chr	229354	51b4ce6294294c466645af51c0293a1b815426fc	Human cytomegalovirus (AD 169)
humdyst.chr	38770	62cbac874cebd290a29cea2445099e318dc71910	Human dystrophin gene (Chr. X)
humghcs.chr	66495	47370ee77b896fb408365f4f826397d9fcfe1dd0	Human growth hormone, etc. (Chr. 17)
humhbb.chr	73309	ba15933f8acab487650d4bb497efd035bd62cca0	Human beta globin region (Chr. 11)
humhdab.chr	58864	352eaa6bff6a5f9b65b1e57a703af29d119aa0e8	Human contig seq. of 3 cosmids.
humprtb.chr	56737	884576c0a13da695b46d48473dd3c792121ed586	Human hypoxanthine phosphoribosyltransferase
mpomtcg.chr	186609	63c37ecacd9879d83bc1931be2ab731cb19ee4d3	Marchantia polymorpha mitochondrium
mtpacga.chr	100314	b4be16f9be6ca78fc0547220306b0cec9ff9c02a	Podospora anserina mitochondrium
vaccg.chr	191737	0ed33634e294a416e151724de4b08f6637b55829	Vaccinia virus

Other Files (large Canterbury corpus)

File	Size	SHA-1 sum	
E.coli	4638690	bea54298e17d5ef86ddb75ac71b5b74fadf2cb7d	DNA of E.coli bacterium
pi.txt	1000000	e995509affabd68e36d0f8f4436cbc2b7541dee5	Decimal expansion of π
random.txt	100000	231f68a1c6d7cee7f1dbb1a5b66b67aa0be0f225	Uniformly distributed symbols

Table A.4: Compression results of selected algorithms, in bits per symbol, on the files of the Canterbury corpus (Arnold and Bell, 1997).
The compression rates were rounded to 3 decimal digits in the final step. Each table cell is shaded to indicate how good the compression rate is relative to that of other compressors on the same file.

worse ← → better

	Method	alice29.txt	asyoulik.txt	cp.html	fields.c	grammar.lsp	kennedy.xls	lcet10.txt	plrabn12.txt	ptt5 (pic)	sum	xargs.1
	MTF	5.072	5.253	5.560	5.416	5.205	**2.951**	5.057	4.992	1.377	**5.218**	5.419
	CRP	4.573	4.815	**5.260**	5.071	4.769	3.575	4.671	4.533	**1.212**	5.358	5.021
	Polya	**4.571**	**4.813**	5.269	**5.049**	**4.730**	3.580	**4.671**	**4.533**	1.225	5.357	**4.983**
	compress	3.274	3.514	3.680	3.562	3.898	2.412	3.058	3.378	0.970	4.205	4.427
	LZW	3.164	3.400	3.537	3.410	3.696	2.414	2.955	3.186	0.937	4.035	4.238
	pkzip	2.892	3.148	2.614	2.305	2.851	1.646	2.744	3.263	0.846	2.711	3.486
	gzip	2.850	3.120	2.593	2.244	**2.653**	1.629	2.707	3.225	0.816	2.671	**3.308**
	7zip	2.554	2.849	2.508	2.192	2.842	**0.400**	2.242	**2.746**	0.679	1.990	3.482
	lzip	**2.551**	**2.848**	**2.478**	**2.152**	2.709	0.409	**2.233**	2.746	**0.618**	**1.982**	3.369
	bzip2	2.272	2.529	2.479	2.180	2.758	1.012	2.019	2.417	0.776	2.701	3.335
	DMC	2.385	2.643	2.687	2.401	2.844	1.414	2.128	2.483	0.817	3.034	3.505
	CSE	2.192	2.493	2.555	2.276	2.750	**0.891**	1.928	2.283	**0.772**	3.024	3.494
	CTW	**2.075**	**2.322**	**2.307**	**1.990**	**2.384**	1.009	**1.832**	**2.185**	0.796	**2.571**	**2.962**
PPM, $\alpha=0$, $\beta=0.5$	PPMD 1	4.573	4.815	5.263	5.077	4.773	3.576	4.672	4.533	1.212	5.365	5.027
	PPMD 2	3.472	3.480	3.760	3.374	3.479	2.790	3.527	3.384	0.849	3.772	3.923
	PPMD 3	2.694	2.777	2.679	2.381	2.634	1.701	2.738	2.804	0.824	3.052	3.191
	PPMD 4	2.273	2.486	2.344	2.118	2.421	1.579	2.160	2.408	0.821	2.800	2.969
	PPMD 5	**2.177**	**2.431**	**2.288**	**2.063**	**2.369**	1.544	1.952	**2.293**	0.822	2.742	**2.952**
	PPMD 6	2.179	2.460	2.291	2.084	2.397	**1.474**	**1.928**	2.312	**0.815**	2.719	2.966
	PPMD 7	2.209	2.500	2.303	2.085	2.414	1.539	1.954	2.370	0.816	2.719	2.983
	PPMD 8	2.240	2.532	2.317	2.079	2.427	1.589	1.990	2.425	0.820	**2.718**	2.998
	PPMD 9	2.265	2.551	2.331	2.090	2.427	1.608	2.023	2.466	0.820	2.723	3.005
	PPMD 10	2.285	2.563	2.345	2.101	2.434	1.615	2.051	2.492	0.823	2.724	3.011
PPM, $\alpha=1$, $\beta=0$	PPMA 1	4.573	4.814	5.264	5.083	4.807	3.576	4.671	4.533	1.213	5.381	5.051
	PPMA 2	3.474	3.479	3.824	3.412	3.578	2.792	3.528	3.383	**0.858**	3.939	4.027
	PPMA 3	2.704	2.791	2.838	2.480	2.784	2.115	2.745	2.804	0.862	3.296	3.380
	PPMA 4	2.309	2.535	2.546	2.255	2.610	1.995	2.181	2.416	0.877	3.151	3.214
	PPMA 5	**2.243**	**2.524**	**2.505**	**2.216**	**2.580**	1.969	1.998	**2.326**	0.885	3.140	**3.208**
	:	:	:	:	:	:	:	:	:	:	:	:

	Method	alice29.txt	asyoulik.txt	cp.html	fields.c	grammar.lsp	kennedy.xls	lcet10.txt	plrabn12.txt	ptt5 (pic)	sum	xargs.1
PPM, $\alpha=0$, $\beta=-0.25$	PPMA 6	2.264	2.578	2.512	2.244	2.610	**1.906**	**1.996**	2.378	0.879	**3.122**	3.221
	PPMA 7	2.307	2.631	2.526	2.251	2.632	2.229	2.040	2.464	0.882	3.126	3.240
	PPMA 8	2.349	2.672	2.541	2.249	2.647	2.407	2.090	2.541	0.886	3.138	3.251
	PPMA 9	2.376	2.692	2.557	2.261	2.649	2.439	2.129	2.592	0.886	3.154	3.261
	PPMA 10	2.397	2.704	2.571	2.272	2.655	2.447	2.160	2.622	0.889	3.158	3.267
	PPME 1	4.573	4.815	5.263	5.077	4.773	3.576	4.672	4.533	1.212	5.365	5.027
	PPME 2	3.472	3.480	3.762	3.377	3.485	2.790	3.527	3.384	0.849	3.777	3.931
	PPME 3	2.695	2.778	2.696	2.379	2.647	1.702	2.739	2.804	**0.827**	3.063	3.217
	PPME 4	2.276	2.491	2.372	2.107	2.429	1.579	2.160	2.409	0.832	2.807	3.005
	PPME 5	**2.187**	**2.452**	2.314	2.036	**2.361**	1.569	1.953	**2.298**	0.839	2.749	**2.985**
	PPME 6	2.194	2.496	**2.309**	2.043	2.376	**1.498**	**1.930**	2.327	0.834	2.718	2.992
	PPME 7	2.224	2.543	2.314	2.033	2.382	1.563	1.954	2.394	0.836	2.710	3.009
	PPME 8	2.256	2.580	2.321	**2.015**	2.386	1.617	1.988	2.460	0.840	**2.705**	3.019
	PPME 9	2.279	2.604	2.330	2.017	2.382	1.637	2.019	2.510	0.838	2.709	3.024
	PPME 10	2.298	2.616	2.338	2.020	2.382	1.643	2.044	2.541	0.840	2.707	3.026
BPPM, $\alpha=0$, $\beta=0.5$	BM-D 1	4.573	4.815	5.260	5.071	4.769	3.575	4.671	4.533	1.212	5.358	5.021
	BM-D 2	3.467	3.473	3.750	3.359	3.459	2.789	3.524	3.382	0.849	3.783	3.899
	BM-D 3	2.680	2.762	2.682	2.352	2.617	1.810	2.729	2.795	**0.830**	3.059	3.170
	BM-D 4	2.259	2.475	2.352	2.065	2.395	**1.764**	2.146	2.392	0.836	2.804	2.962
	BM-D 5	**2.180**	**2.453**	**2.293**	**1.991**	**2.350**	1.820	1.941	**2.284**	0.845	2.744	**2.956**
	BM-D 6	2.211	2.532	2.303	2.004	2.384	1.829	**1.931**	2.333	0.844	**2.732**	2.988
	BM-D 7	2.277	2.629	2.326	2.011	2.412	1.902	1.981	2.441	0.851	2.754	3.030
	BM-D 8	2.351	2.714	2.353	2.020	2.449	1.954	2.052	2.558	0.859	2.771	3.070
	BM-D 9	2.415	2.776	2.380	2.045	2.470	2.030	2.121	2.659	0.863	2.785	3.106
	BM-D 10	2.471	2.820	2.405	2.071	2.494	2.106	2.185	2.734	0.870	2.800	3.132
BPPM, $\alpha=0.5$, $\beta=0.75$	BM-J 1	4.574	4.816	5.264	5.078	4.782	3.575	4.672	4.534	1.212	5.357	5.032
	BM-J 2	3.470	3.478	3.751	3.384	3.483	2.790	3.525	3.383	0.848	3.781	3.908
	BM-J 3	2.681	2.764	2.658	2.396	2.649	1.767	2.731	2.794	0.817	3.049	3.164
	BM-J 4	2.242	2.447	2.303	2.096	2.399	1.677	2.142	2.381	0.804	2.766	2.915
	BM-J 5	2.113	2.351	2.209	1.982	2.307	1.687	1.910	2.235	0.798	2.660	2.848
	BM-J 6	2.073	2.336	2.181	1.944	2.288	**1.650**	1.847	**2.209**	0.788	2.602	2.826
	BM-J 7	**2.066**	**2.344**	2.167	1.905	2.270	1.674	**1.832**	2.222	0.787	2.577	2.816
	BM-J 8	2.069	2.354	2.159	1.873	2.264	1.677	1.833	2.244	0.789	2.552	**2.816**
	BM-J 9	2.074	2.363	2.157	1.860	2.249	1.707	1.838	2.262	**0.786**	2.535	2.816
	BM-J 10	2.082	2.369	**2.156**	1.850	2.245	1.736	1.847	2.277	0.787	2.525	2.816
	:	:	:	:	:	:	:	:	:	:	:	:

	Method	alice29.txt	asyoulik.txt	cp.html	fields.c	grammar.lsp	kennedy.xls	lcet10.txt	plrabn12.txt	ptt5 (pic)	sum	xargs.1
BPPM, $\alpha=0.5$, $\beta=0.85$	BM-J 11	2.090	2.375	2.158	1.847	**2.242**	1.765	1.854	2.287	0.790	2.514	2.818
	BM-J 12	2.095	2.378	2.160	**1.846**	2.245	1.823	1.860	2.294	0.793	2.506	2.820
	BM-J 13	2.100	2.381	2.161	1.845	2.247	1.883	1.866	2.299	0.795	2.498	2.822
	BM-J 14	2.104	2.383	2.164	1.848	2.249	1.883	1.870	2.302	0.797	2.493	2.826
	BM-J 15	2.107	2.385	2.166	1.849	2.251	1.883	1.874	2.303	0.799	2.491	2.828
	BM-J 20	2.116	2.388	2.180	1.857	2.260	1.883	1.884	2.306	0.807	**2.482**	2.837
	BM-K 1	4.574	4.816	5.266	5.083	4.790	3.575	4.672	4.534	1.212	5.357	5.040
	BM-K 2	3.472	3.481	3.760	3.405	3.513	2.790	3.527	3.384	0.848	3.794	3.933
	BM-K 3	2.687	2.771	2.676	2.435	2.702	1.754	2.735	2.796	0.816	3.076	3.210
	BM-K 4	2.253	2.457	2.331	2.146	2.464	1.651	2.149	2.386	0.801	2.798	2.969
	BM-K 5	2.122	2.356	2.239	2.036	2.371	1.648	1.919	2.239	0.793	2.691	2.903
	BM-K 6	2.077	2.331	2.209	1.997	2.350	1.598	1.852	**2.205**	0.781	2.629	2.877
	BM-K 7	2.063	**2.328**	2.192	1.955	2.331	1.608	1.831	2.207	0.780	2.600	2.863
	BM-K 8	2.057	2.329	2.181	1.921	2.318	**1.597**	1.824	2.215	0.781	2.570	2.858
	BM-K 9	**2.054**	2.330	2.174	1.904	2.300	1.614	**1.821**	2.222	0.777	2.549	2.852
	BM-K 10	2.056	2.331	2.169	1.892	2.290	1.630	1.822	2.228	**0.777**	2.535	2.850
	BM-K 11	2.058	2.333	2.167	1.883	2.285	1.647	1.823	2.232	0.778	2.519	2.846
	BM-K 12	2.059	2.334	2.165	1.878	2.283	1.683	1.824	2.235	0.779	2.508	2.846
	BM-K 13	2.061	2.334	2.162	1.874	2.281	1.717	1.825	2.237	0.781	2.498	**2.845**
	BM-K 14	2.062	2.335	2.162	1.872	2.281	1.717	1.826	2.238	0.782	2.491	2.846
	BM-K 15	2.063	2.335	2.161	1.870	2.279	1.717	1.826	2.239	0.783	2.486	2.846
	BM-K 20	2.067	2.336	2.162	1.864	2.279	1.717	1.829	2.240	0.786	2.468	2.850
	BM-K 25	2.068	2.337	2.165	**1.863**	2.281	1.717	1.830	2.240	0.790	2.460	2.852
	BM-K 30	2.068	2.337	2.169	1.865	2.281	1.717	1.830	2.240	0.794	**2.457**	2.852
BPPM, 7 depth-dep. parameters	DDA 1	4.572	4.814	5.257	5.065	4.762	3.575	4.671	4.533	1.212	5.367	5.013
	DDA 2	3.467	3.474	3.742	3.360	3.459	2.790	3.524	3.382	0.848	3.784	3.884
	DDA 3	2.679	2.761	2.654	2.382	2.638	1.768	2.729	2.793	0.818	3.064	3.153
	DDA 4	2.241	2.446	2.289	2.063	2.378	1.672	2.139	2.382	0.807	2.765	2.886
	DDA 5	2.113	2.349	2.207	1.968	2.296	1.667	1.910	2.236	0.800	2.672	2.835
	DDA 6	2.070	2.326	2.180	1.933	2.283	1.612	1.845	2.206	0.788	2.613	2.814
	DDA 7	2.056	2.320	2.178	1.912	2.273	1.612	1.826	**2.201**	0.786	2.603	2.814
	DDA 8	2.049	2.317	2.173	1.889	2.268	**1.591**	1.818	2.203	0.787	2.582	2.812
	DDA 9	2.045	2.315	2.170	1.881	2.255	1.598	1.813	2.203	0.782	2.568	2.811
	DDA 10	2.043	2.314	2.168	1.873	2.249	1.604	1.812	2.203	**0.781**	2.558	2.809
	DDA 15	2.041	2.313	2.160	1.857	2.240	1.640	1.804	2.204	0.784	2.519	**2.805**
	:	:	:	:	:	:	:	:	:	:	:	:

	Method	alice29.txt	asyoulik.txt	cp.html	fields.c	grammar.lsp	kennedy.xls	lcet10.txt	plrabn12.txt	ptt5 (pic)	sum	xargs.1
BPPM, 4 fanout-dep. parameters	DDA 20	2.041	2.312	2.156	1.848	2.236	1.640	1.802	2.204	0.783	2.503	2.805
	DDA 25	2.041	2.312	2.153	1.843	**2.234**	1.640	1.801	2.203	0.782	2.491	2.805
	DDA 30	**2.040**	**2.312**	**2.152**	**1.840**	2.234	1.640	**1.800**	2.203	0.783	**2.485**	2.805
	FDA 1	4.574	4.816	5.266	5.083	4.790	3.575	4.672	4.534	1.212	5.357	5.042
	FDA 2	3.473	3.483	3.761	3.419	3.519	2.793	3.527	3.384	0.848	3.799	3.946
	FDA 3	2.690	2.773	2.677	2.445	2.690	1.754	2.736	2.798	0.814	3.085	3.221
	FDA 4	2.252	2.452	2.319	2.137	2.434	1.652	2.151	2.387	0.798	2.788	2.954
	FDA 5	2.114	2.343	2.214	2.015	2.331	1.650	1.913	2.232	0.790	2.669	2.875
	FDA 6	2.065	2.315	2.178	1.968	2.307	1.601	1.840	2.192	0.778	2.600	2.843
	FDA 7	2.048	**2.312**	2.158	1.923	2.283	1.612	1.815	**2.191**	0.776	2.567	2.828
	FDA 8	2.042	2.315	2.147	1.885	2.270	**1.601**	1.807	2.200	0.776	2.534	2.822
	FDA 9	**2.041**	2.318	2.141	1.866	2.253	1.618	**1.804**	2.210	0.772	2.513	2.818
	FDA 10	2.045	2.322	2.137	1.853	2.247	1.635	1.806	2.219	**0.772**	2.498	2.818
	FDA 11	2.049	2.325	2.136	1.847	**2.242**	1.652	1.809	2.226	0.773	2.484	**2.816**
	FDA 12	2.053	2.328	2.136	1.843	2.242	1.689	1.812	2.231	0.775	2.473	2.818
	FDA 13	2.056	2.329	**2.135**	1.840	2.245	1.725	1.815	2.235	0.776	2.465	2.818
	FDA 14	2.059	2.331	2.136	**1.840**	2.245	1.725	1.818	2.237	0.777	2.458	2.820
	FDA 15	2.061	2.332	2.137	1.841	2.247	1.725	1.820	2.238	0.778	2.456	2.824
	FDA 20	2.069	2.335	2.145	1.845	2.255	1.725	1.828	2.241	0.781	2.445	2.831
	FDA 25	2.072	2.336	2.155	1.852	2.262	1.725	1.831	2.241	0.786	**2.443**	2.835
	FDA 30	2.073	2.337	2.163	1.857	2.266	1.725	1.833	2.241	0.791	2.444	2.835
BPPM, 6 fanout-dep. parameters	FDB 1	4.574	4.816	5.266	5.083	4.790	3.575	4.672	4.534	1.212	5.357	5.042
	FDB 2	3.474	3.484	3.761	3.422	3.519	2.793	3.527	3.384	0.848	3.801	3.948
	FDB 3	2.691	2.774	2.678	2.450	2.696	1.754	2.737	2.798	0.813	3.090	3.227
	FDB 4	2.253	2.453	2.322	2.145	2.442	1.653	2.151	2.388	0.797	2.795	2.964
	FDB 5	2.115	2.344	2.219	2.024	2.341	1.652	1.914	2.232	0.789	2.677	2.886
	FDB 6	2.066	2.315	2.183	1.978	2.318	**1.604**	1.842	2.192	0.777	2.609	2.852
	FDB 7	2.048	**2.311**	2.164	1.932	2.294	1.616	1.816	**2.189**	0.775	2.576	2.837
	FDB 8	2.042	2.312	2.152	1.894	2.281	1.605	1.807	2.197	0.775	2.543	2.831
	FDB 9	**2.040**	2.314	2.145	1.876	2.264	1.623	**1.804**	2.205	0.771	2.521	2.828
	FDB 10	2.042	2.317	2.141	1.862	2.255	1.640	1.805	2.213	**0.771**	2.506	2.826
	FDB 11	2.045	2.319	2.139	1.854	2.249	1.657	1.806	2.218	0.772	2.490	**2.824**
	FDB 12	2.048	2.322	2.138	1.850	**2.249**	1.695	1.808	2.223	0.773	2.479	2.824
	FDB 13	2.051	2.323	**2.136**	1.846	2.249	1.733	1.810	2.226	0.774	2.470	2.824
	FDB 14	2.053	2.324	2.136	1.845	2.249	1.733	1.813	2.228	0.775	2.464	2.826
	FDB 15	2.055	2.325	2.137	**1.845**	2.251	1.733	1.814	2.229	0.777	2.460	2.828
	:	:	:	:	:	:	:	:	:	:	:	:

Method	alice29.txt	asyoulik.txt	cp.html	fields.c	grammar.lsp	kennedy.xls	lcet10.txt	plrabn12.txt	ptt5 (pic)	sum	xargs.1
FDB 20	2.062	2.328	2.143	1.846	2.257	1.733	1.820	2.231	0.779	2.448	2.835
FDB 25	2.064	2.329	2.150	1.851	2.262	1.733	1.823	2.231	0.783	**2.444**	2.839
FDB 30	2.065	2.329	2.157	1.855	2.264	1.733	1.824	2.232	0.788	2.444	2.839
SM-0	2.680	2.940	2.752	2.383	2.763	2.475	2.467	2.880	?	2.755	3.280
SM-JG	2.049	2.314	2.154	**1.829**	**2.234**	1.615	1.806	2.215	?	2.450	**2.812**
ppmz2	2.059	2.309	2.158	1.896	2.300	1.373	1.794	**2.194**	**0.754**	2.538	2.850
PPMII	**2.033**	**2.308**	**2.139**	1.845	2.268	**1.168**	**1.791**	2.202	0.757	**2.327**	2.852
lpaq1	1.955	2.197	2.087	1.832	2.298	0.294	1.683	2.106	0.680	2.221	2.867
paq8l	**1.843**	**2.112**	**1.917**	**1.608**	**2.081**	**0.091**	**1.573**	**2.025**	**0.351**	**1.576**	**2.693**

See Table A.1 for descriptions of the compression methods used here.

Table A.5: Compression results of selected algorithms in bits per symbol, on the files of the Calgary corpus (Bell et al., 1990). The file `pic` of the Calgary corpus is identical to file `ptt5` of the Canterbury corpus (see Table A.4), and therefore omitted here.

	Method	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	progc	progl	progp	trans
	MTF	5.625	4.946	5.027	**5.507**	5.468	6.059	**6.166**	5.256	4.984	5.533	4.891	5.170	**5.502**
	CRP	5.209	4.528	**4.794**	5.660	**5.193**	**6.000**	6.267	**4.999**	4.612	5.221	4.782	4.886	5.543
	Polya	**5.206**	**4.528**	4.823	5.665	5.204	6.035	6.272	5.025	**4.608**	**5.213**	**4.778**	**4.880**	5.541
	compress	3.346	3.455	3.284	6.076	3.864	5.226	4.170	3.774	3.519	3.866	3.031	3.112	3.265
	LZW	3.223	3.172	3.054	5.856	3.630	5.015	4.041	3.640	3.382	3.707	2.926	2.984	3.138
	pkzip	2.583	3.289	2.739	5.392	3.100	3.875	2.656	2.819	2.910	2.716	1.824	1.840	1.691
	gzip	2.509	3.250	2.700	5.345	3.063	3.837	2.628	2.789	2.887	2.677	1.804	1.811	1.610
	7z	2.208	**2.717**	2.225	**4.153**	2.533	3.519	2.001	2.604	2.656	2.543	1.680	1.687	1.441
	lzip	**2.199**	2.717	**2.224**	4.185	**2.521**	**3.506**	**1.991**	**2.598**	**2.655**	**2.532**	**1.666**	**1.671**	**1.420**
	bzip2	1.975	2.420	2.062	**4.447**	2.516	4.013	2.478	2.492	2.437	2.533	1.740	1.735	1.528
	DMC	2.196	2.595	2.296	4.801	2.773	4.121	2.765	2.730	2.587	2.754	1.987	1.996	1.917
	CSE	1.975	2.268	1.977	5.354	2.525	4.462	2.711	2.540	2.412	2.604	1.712	1.778	1.598
	CTW	**1.833**	**2.180**	**1.891**	4.532	**2.350**	**3.721**	**2.398**	**2.291**	**2.229**	**2.337**	**1.647**	**1.679**	**1.443**
PPM, $\alpha=1$, $\beta=0$	PPMA 1	5.209	4.528	4.794	5.669	5.193	6.042	6.270	5.001	4.613	5.223	4.783	4.887	5.544
	PPMA 2	3.455	3.601	3.778	**5.068**	4.163	5.022	4.202	3.834	3.626	3.877	3.320	3.373	3.497
	PPMA 3	2.664	2.901	2.898	5.367	3.307	4.627	3.289	2.958	2.884	2.984	2.407	2.313	2.386
	PPMA 4	2.153	2.469	2.287	6.000	2.757	4.525	2.984	2.562	2.484	2.651	1.939	1.909	1.813
	PPMA 5	2.002	**2.333**	2.068	6.083	2.581	**4.517**	2.811	**2.480**	**2.408**	**2.593**	1.816	1.846	1.643
	:	:	:	:	:	:	:	:	:	:	:	:	:	:

	Method	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	progc	progl	progp	trans
PPM, $\alpha=0$, $\beta=0.5$	PPMA 6	**1.997**	2.360	**2.042**	6.082	**2.576**	4.527	2.760	2.501	2.434	2.608	1.782	**1.843**	1.608
	PPMA 7	2.024	2.427	2.070	6.083	2.605	4.537	2.759	2.530	2.479	2.626	1.771	1.851	1.590
	PPMA 8	2.049	2.496	2.110	6.080	2.632	4.544	2.758	2.551	2.515	2.644	**1.766**	1.861	**1.585**
	PPMA 9	2.070	2.544	2.145	6.081	2.652	4.550	**2.750**	2.566	2.538	2.657	1.767	1.869	1.592
	PPMA 10	2.089	2.576	2.175	6.081	2.666	4.555	2.755	2.579	2.552	2.668	1.777	1.878	1.598
	PPMD 1	5.210	4.529	4.795	5.666	5.193	6.014	6.270	5.001	4.613	5.224	4.783	4.887	5.545
	PPMD 2	3.451	3.601	3.775	4.652	4.150	4.536	4.092	3.818	3.619	3.841	3.316	3.356	3.487
	PPMD 3	2.631	2.896	2.882	**4.589**	3.248	3.962	3.080	2.902	2.856	2.878	2.388	2.261	2.356
	PPMD 4	2.088	2.452	2.253	4.714	2.634	3.807	2.725	2.460	2.422	2.485	1.889	1.828	1.750
	PPMD 5	1.910	2.291	2.014	4.773	2.411	**3.778**	2.512	**2.346**	**2.315**	**2.402**	1.746	1.747	1.555
	PPMD 6	**1.884**	**2.291**	**1.969**	4.775	**2.379**	3.784	2.449	2.351	2.322	2.404	1.701	1.732	1.507
	PPMD 7	1.896	2.336	1.980	4.775	2.393	3.793	2.439	2.373	2.354	2.413	1.680	**1.732**	1.483
	PPMD 8	1.916	2.388	2.009	4.773	2.413	3.798	2.434	2.390	2.384	2.428	1.670	1.737	**1.473**
	PPMD 9	1.935	2.430	2.038	4.774	2.431	3.804	**2.421**	2.405	2.406	2.442	**1.667**	1.740	1.477
	PPMD 10	1.953	2.458	2.064	4.773	2.443	3.808	2.424	2.418	2.421	2.453	1.675	1.747	1.480
PPM, $\alpha=0$, $\beta=-0.25$	PPME 1	5.210	4.529	4.795	5.666	5.193	6.014	6.270	5.001	4.613	5.224	4.783	4.887	5.545
	PPME 2	3.452	3.601	3.775	4.655	4.150	4.550	4.093	3.819	3.620	3.843	3.316	3.356	3.487
	PPME 3	2.634	2.897	2.883	**4.581**	3.251	4.004	3.086	2.908	2.859	2.884	2.390	2.262	2.354
	PPME 4	2.096	2.455	2.256	4.743	2.645	3.861	2.733	2.470	2.431	2.496	1.889	1.823	1.738
	PPME 5	1.913	**2.301**	2.019	4.851	2.428	**3.835**	2.521	**2.357**	**2.333**	2.411	1.740	1.730	1.527
	PPME 6	**1.879**	2.311	**1.974**	4.855	**2.399**	3.839	2.451	2.361	2.346	**2.407**	1.684	1.702	1.465
	PPME 7	1.887	2.366	1.983	4.856	2.412	3.846	2.432	2.379	2.379	2.411	1.653	1.689	1.429
	PPME 8	1.903	2.427	2.008	4.853	2.430	3.850	2.419	2.393	2.409	2.421	1.634	1.683	1.408
	PPME 9	1.917	2.475	2.034	4.855	2.446	3.855	2.400	2.406	2.430	2.429	**1.623**	1.676	1.403
	PPME 10	1.928	2.507	2.056	4.854	2.455	3.858	**2.398**	2.415	2.443	2.435	1.623	1.673	**1.398**
	PPME 11	1.937	2.529	2.074	4.853	2.462	3.860	2.399	2.423	2.453	2.439	1.628	1.671	1.400
	PPME 12	1.944	2.543	2.089	4.853	2.469	3.863	2.401	2.428	2.459	2.442	1.632	1.669	1.404
	PPME 13	1.951	2.550	2.100	4.853	2.473	3.866	2.402	2.432	2.464	2.445	1.634	**1.667**	1.409
BPPM, $\alpha=0$, $\beta=0.5$	BM-D 1	5.209	4.528	4.794	5.660	5.193	6.000	6.267	4.999	4.612	5.221	4.782	4.886	5.543
	BM-D 2	3.444	3.599	3.773	4.670	4.145	4.547	4.086	3.807	3.611	3.830	3.310	3.347	3.477
	BM-D 3	2.618	2.891	2.876	**4.603**	3.243	3.990	3.075	2.885	2.840	2.865	2.375	2.247	2.335
	BM-D 4	2.079	2.443	2.247	4.894	2.638	3.849	2.716	2.448	2.412	2.476	1.872	1.800	1.710
	BM-D 5	1.898	**2.294**	2.012	4.971	2.424	**3.838**	2.508	**2.351**	**2.323**	**2.399**	1.717	1.702	1.486
	BM-D 6	**1.873**	2.322	**1.980**	5.018	**2.406**	3.865	**2.456**	2.381	2.361	2.417	1.666	**1.680**	1.414
	BM-D 7	1.901	2.412	2.015	5.052	2.440	3.892	2.459	2.433	2.432	2.449	1.649	1.682	1.378
	BM-D 8	1.940	2.518	2.074	5.052	2.485	3.912	2.472	2.485	2.505	2.488	**1.646**	1.690	**1.363**
	BM-D 9	1.980	2.614	2.137	5.053	2.526	3.931	2.473	2.531	2.566	2.526	1.652	1.700	1.365
	BM-D 10	2.018	2.692	2.197	5.053	2.559	3.946	2.492	2.569	2.614	2.559	1.670	1.713	1.372
	:	:	:	:	:	:	:	:	:	:	:	:	:	:

	Method	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	progc	progl	progp	trans
BPPM, $\alpha=0.5$, $\beta=0.75$	BM-J 1	5.210	4.529	4.795	5.659	5.193	5.997	6.267	5.001	4.613	5.224	4.784	4.888	5.545
	BM-J 2	3.449	3.600	3.774	4.620	4.147	4.492	4.083	3.814	3.615	3.839	3.316	3.355	3.486
	BM-J 3	2.616	2.890	2.877	**4.471**	3.239	3.881	3.069	2.892	2.838	2.872	2.384	2.268	2.360
	BM-J 4	2.059	2.430	2.239	4.533	2.615	3.684	2.686	2.433	2.388	2.456	1.878	1.817	1.750
	BM-J 5	1.853	2.243	1.977	4.532	2.361	3.630	2.442	2.283	2.242	2.327	1.701	1.695	1.521
	BM-J 6	1.787	**2.202**	1.896	4.536	2.287	3.622	2.351	2.245	2.207	2.284	1.616	1.639	1.426
	BM-J 7	1.762	2.206	1.870	4.545	2.261	3.622	2.309	2.232	**2.202**	2.260	1.561	1.604	1.361
	BM-J 8	1.753	2.225	**1.865**	4.542	2.252	**3.621**	2.282	**2.227**	2.208	2.251	1.520	1.576	1.317
	BM-J 9	1.749	2.243	1.867	4.542	2.248	3.624	2.251	2.227	2.215	2.247	1.493	1.551	1.289
	BM-J 10	**1.749**	2.258	1.873	4.541	2.246	3.626	2.240	2.229	2.221	2.246	1.478	1.534	1.266
	BM-J 15	1.761	2.292	1.903	4.541	2.246	3.641	2.221	2.239	2.241	2.249	**1.455**	1.484	1.229
	BM-J 20	1.775	2.296	1.916	4.542	2.249	3.653	2.221	2.246	2.248	2.259	1.458	**1.472**	1.229
BPPM, $\alpha=0.5$, $\beta=0.85$	BM-K 1	5.210	4.529	4.795	5.659	5.193	5.997	6.267	5.002	4.613	5.226	4.785	4.889	5.545
	BM-K 2	3.453	3.600	3.775	4.613	4.148	4.500	4.088	3.822	3.618	3.849	3.321	3.363	3.492
	BM-K 3	2.624	2.892	2.882	**4.450**	3.247	3.892	3.086	2.911	2.847	2.896	2.398	2.289	2.380
	BM-K 4	2.074	2.434	2.247	4.465	2.631	3.692	2.708	2.461	2.403	2.489	1.901	1.850	1.785
	BM-K 5	1.872	2.244	1.986	4.455	2.381	3.634	2.468	2.313	2.256	2.361	1.730	1.732	1.566
	BM-K 6	1.806	2.197	1.902	4.454	2.304	3.622	2.377	2.271	2.215	2.315	1.646	1.677	1.475
	BM-K 7	1.778	**2.191**	1.870	4.459	2.273	3.618	2.332	2.253	2.204	2.287	1.590	1.640	1.412
	BM-K 8	1.763	2.198	1.858	4.455	2.258	3.614	2.301	2.241	**2.201**	2.273	1.548	1.609	1.368
	BM-K 9	1.755	2.205	1.853	4.455	2.248	3.614	2.267	2.236	2.201	2.264	1.518	1.582	1.338
	BM-K 10	1.750	2.212	**1.852**	4.454	2.241	**3.613**	2.252	2.233	2.202	2.258	1.500	1.562	1.312
	BM-K 15	**1.739**	2.226	1.857	4.452	2.225	3.618	2.217	2.228	2.206	2.245	1.460	1.496	1.256
	BM-K 20	1.741	2.229	1.862	4.452	**2.220**	3.624	2.205	**2.228**	2.209	2.246	1.449	1.472	1.237
	BM-K 25	1.744	2.229	1.864	4.453	2.218	3.628	**2.201**	2.228	2.209	2.247	1.446	1.459	1.230
	BM-K 30	1.745	2.229	1.865	4.453	2.218	3.631	2.202	2.228	2.210	2.248	**1.446**	**1.454**	**1.226**
BPPM, 7 depth-dep. parameters	DDA 1	5.208	4.528	4.794	5.663	5.192	6.015	6.268	4.997	4.610	5.218	4.780	4.884	5.542
	DDA 2	3.444	3.599	3.772	4.653	4.145	4.529	4.083	3.805	3.610	3.827	3.309	3.346	3.477
	DDA 3	2.612	2.889	2.876	4.512	3.238	3.932	3.074	2.886	2.836	2.866	2.380	2.262	2.355
	DDA 4	2.053	2.431	2.237	4.574	2.612	3.728	2.685	2.421	2.383	2.441	1.867	1.801	1.729
	DDA 5	1.854	2.243	1.977	4.565	2.366	3.680	2.455	2.280	2.241	2.322	1.701	1.692	1.520
	DDA 6	1.787	2.198	1.895	4.566	2.290	**3.670**	2.365	2.239	2.202	2.280	1.618	1.639	1.430
	DDA 7	1.775	**2.187**	1.866	4.568	2.274	3.673	2.338	2.232	2.194	2.265	1.581	1.620	1.401
	DDA 8	1.766	2.188	1.854	4.565	2.263	3.671	2.316	2.224	2.192	2.256	1.549	1.599	1.373
	DDA 9	1.760	2.190	1.848	4.564	2.257	3.671	2.289	2.221	2.190	2.249	1.526	1.578	1.354
	DDA 10	1.756	2.191	1.844	4.563	2.251	**3.670**	2.278	2.218	2.189	2.245	1.514	1.563	1.337
	DDA 15	1.739	2.193	1.838	4.561	2.236	3.672	2.247	2.210	2.186	2.232	1.483	1.509	1.297
	DDA 20	1.735	2.193	1.837	4.560	2.227	3.674	2.232	2.207	2.186	2.228	1.469	1.484	1.276
	DDA 25	1.732	2.193	1.836	**4.560**	2.222	3.676	2.225	2.205	2.185	2.226	1.460	1.468	1.262
	DDA 30	1.730	2.193	1.836	4.560	2.218	3.678	2.221	2.204	2.185	2.225	1.455	1.458	1.251
	DDA 35	**1.730**	2.193	**1.836**	4.560	**2.216**	3.680	**2.218**	**2.204**	**2.185**	**2.225**	**1.451**	**1.452**	**1.244**
	:	:	:	:	:	:	:	:	:	:	:	:	:	:

BPPM, 6 fanout-dep. parameters

Method	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	progc	progl	progp	trans
FDB 1	5.211	4.529	4.795	5.659	5.193	5.997	6.267	5.003	4.614	5.226	4.784	4.889	5.545
FDB 2	3.457	3.601	3.775	4.611	4.148	4.495	4.087	3.825	3.621	3.852	3.323	3.367	3.495
FDB 3	2.629	2.893	2.882	**4.437**	3.245	3.880	3.091	2.918	2.853	2.902	2.398	2.300	2.389
FDB 4	2.078	2.434	2.247	4.447	2.625	3.670	2.704	2.460	2.405	2.486	1.898	1.852	1.788
FDB 5	1.868	2.239	1.980	4.439	2.367	3.608	2.457	2.303	2.250	2.347	1.721	1.726	1.556
FDB 6	1.793	2.185	1.891	4.441	2.282	3.595	2.359	2.253	2.203	2.297	1.630	1.664	1.456
FDB 7	1.760	**2.175**	1.856	4.448	2.247	3.590	2.310	2.231	2.188	2.265	1.570	1.622	1.386
FDB 8	1.743	2.181	1.841	4.445	2.230	3.586	2.276	2.219	**2.185**	2.249	1.524	1.588	1.336
FDB 9	1.734	2.189	1.835	4.445	2.220	3.586	2.241	2.213	2.186	2.239	1.492	1.560	1.303
FDB 10	1.729	2.197	**1.834**	4.443	2.213	**3.585**	2.225	2.211	2.187	2.234	1.473	1.538	1.276
FDB 11	1.725	2.204	1.835	4.443	2.208	3.586	2.215	2.210	2.189	2.229	1.461	1.520	1.259
FDB 12	1.722	2.209	1.837	4.442	2.206	3.588	2.208	**2.209**	2.191	2.227	1.452	1.504	1.246
FDB 13	**1.721**	2.213	1.840	4.441	2.203	3.589	2.202	2.210	2.194	2.225	1.445	1.491	1.237
FDB 14	1.722	2.216	1.842	4.441	2.202	3.589	2.197	2.210	2.195	**2.225**	1.440	1.483	1.229
FDB 15	1.722	2.218	1.845	4.442	2.201	3.591	2.192	2.211	2.197	2.225	1.437	1.475	1.223
FDB 20	1.729	2.222	1.853	4.442	**2.199**	3.597	**2.186**	2.214	2.201	2.230	**1.433**	1.456	**1.213**
FDB 25	1.735	2.222	1.857	4.442	2.200	3.600	2.187	2.217	2.203	2.234	1.436	1.449	1.215
FDB 30	1.739	2.222	1.859	4.442	2.202	3.603	2.190	2.218	2.204	2.237	1.440	**1.448**	1.217
SM-0	2.232	2.874	2.505	5.049	2.789	4.105	2.769	2.770	2.787	2.779	2.018	1.961	1.767
SM-JG	1.734	2.206	1.842	4.516	2.215	3.650	2.205	2.209	2.190	2.226	1.443	1.439	1.231
ppmz2	1.718	2.188	1.839	4.578	2.205	3.667	2.241	2.212	2.185	2.257	1.447	1.449	1.214
PPMII	1.726	2.185	1.827	4.317	2.188	3.506	2.160	2.190	2.173	2.198	1.437	1.445	1.222
lpaq1	1.664	2.108	1.715	3.986	2.062	3.494	1.961	2.094	2.080	2.100	1.345	1.362	1.145
paq8l	**1.500**	**2.006**	**1.596**	**3.438**	**1.907**	**2.787**	**1.456**	**1.970**	**1.994**	**1.923**	**1.187**	**1.157**	**0.995**

See Table A.1 for descriptions of the compression methods used here.

Table A.6: Compression results of selected algorithms in bits per symbol, on the files of the DNA corpus by Grumbach and Tahi (1994). Only four symbols {a, g, c, t} occur in these sequences, one for each nucleotide.

	Method	chmpxx.chr	chntxx.chr	hehcmv.chr	humdyst.chr	humghcs.chr	humhbb.chr	humhdab.chr	humprtb.chr	mpomtcg.chr	mtpacga.chr	vaccg.chr
	MTF	1.941	1.984	1.994	1.989	**1.991**	1.989	**1.994**	1.987	1.984	1.969	1.990
	CRP	**1.867**	**1.958**	**1.986**	**1.950**	2.001	**1.969**	1.999	**1.972**	**1.984**	**1.880**	**1.920**
	Polya	1.868	1.958	1.986	1.951	2.002	1.970	2.000	1.973	1.984	1.881	1.920
	compr	2.091	2.187	2.199	2.233	2.194	2.201	2.207	2.202	2.202	2.116	2.136
	LZW	**2.011**	**2.104**	**2.124**	**2.144**	2.102	2.111	2.114	2.110	**2.126**	**2.044**	2.062
	pkzip	2.305	2.379	2.381	2.411	1.874	2.300	2.285	2.313	2.383	2.314	2.287
	gzip	2.221	2.291	2.282	2.377	1.552	2.230	2.209	2.232	2.280	2.232	2.190
	7z	2.080	2.161	2.163	2.224	1.144	2.074	2.018	**2.055**	2.126	2.094	2.066
	lzip	2.095	2.172	2.152	2.226	**1.131**	**2.053**	**2.014**	2.070	2.144	2.092	**2.062**
	bzip2	2.122	2.184	2.168	2.180	1.729	2.148	2.068	2.094	2.170	2.123	2.095
	DMC	1.988	2.090	2.122	2.078	1.901	2.062	2.005	2.017	2.108	2.011	2.018
	CSE	2.014	2.121	2.132	2.151	**1.459**	2.096	2.000	2.029	2.123	2.034	2.027
	CTW	**1.823**	**1.927**	**1.950**	**1.923**	1.882	**1.913**	**1.902**	**1.917**	**1.963**	**1.867**	**1.877**
PPM, $\alpha=1$, $\beta=0$	PPMA 1	1.867	1.957	1.986	1.949	2.001	1.969	1.999	1.972	1.984	1.880	1.919
	PPMA 2	1.846	1.938	1.977	**1.920**	1.939	1.923	1.958	1.937	1.969	1.871	1.916
	PPMA 3	1.844	**1.935**	1.970	1.921	1.935	**1.919**	1.945	**1.928**	**1.965**	**1.871**	1.910
	PPMA 4	**1.843**	1.938	**1.961**	1.939	1.937	1.928	**1.943**	1.931	1.967	1.873	**1.909**
	PPMA 5	1.862	1.953	1.971	1.988	1.954	1.955	1.963	1.956	1.981	1.893	1.921
	PPMA 6	1.913	2.006	2.011	2.094	1.978	2.025	2.016	2.019	2.025	1.946	1.957
	PPMA 7	1.994	2.112	2.104	2.218	1.953	2.140	2.077	2.097	2.128	2.031	2.034
	PPMA 8	2.071	2.221	2.229	2.267	1.805	2.208	2.094	2.123	2.236	2.110	2.131
	PPMA 9	2.117	2.272	2.290	2.273	1.655	2.209	2.084	2.121	2.266	2.155	2.186
	PPMA 10	2.141	2.283	2.300	2.273	**1.590**	2.199	2.080	2.119	2.267	2.168	2.197
PPM, $\alpha=0$, $\beta=0.5$	PPMD 1	1.867	1.958	1.986	1.950	2.001	1.969	1.999	1.972	1.984	1.880	1.920
	PPMD 2	1.847	1.938	1.977	**1.921**	1.940	1.924	1.959	1.938	1.969	**1.872**	1.916
	PPMD 3	**1.845**	**1.936**	1.971	1.925	1.938	**1.921**	**1.948**	**1.931**	**1.966**	1.873	**1.911**
	PPMD 4	1.848	1.943	**1.964**	1.953	1.947	1.936	1.953	1.942	1.971	1.879	1.912
	PPMD 5	1.879	1.968	1.982	2.030	1.982	1.980	1.995	1.988	1.994	1.913	1.933
	PPMD 6	1.961	2.050	2.044	2.194	2.051	2.092	2.099	2.098	2.065	2.000	1.992
	PPMD 7	2.088	2.219	2.195	2.376	2.080	2.278	2.213	2.238	2.234	2.133	2.120
	PPMD 8	2.192	2.369	2.381	2.414	1.939	2.364	2.227	2.262	2.390	2.244	2.266
	PPMD 9	2.246	2.416	2.439	2.404	1.749	2.347	2.204	2.242	2.407	2.294	2.330
	PPMD 10	2.268	2.416	2.434	2.399	1.653	2.324	2.193	2.234	2.397	2.299	2.327
	:	:	:	:	:	:	:	:	:	:	:	:

	Method	chmpxx.chr	chntxx.chr	hehcmv.chr	humdyst.chr	humghcs.chr	humhbb.chr	humhdab.chr	humprtb.chr	mpomtcg.chr	mtpacga.chr	vaccg.chr
PPM, $\alpha=0$, $\beta=-0.25$	PPMD 11	2.275	2.412	2.429	2.398	1.624	2.316	2.194	2.234	2.388	2.295	2.317
	PPMD 12	2.277	2.411	2.427	2.398	**1.622**	2.315	2.199	2.237	2.386	2.292	2.313
	PPMD 13	2.277	2.411	2.427	2.398	1.626	2.315	2.204	2.241	2.385	2.291	2.312
	PPMD 14	2.277	2.411	2.427	2.398	1.634	2.315	2.209	2.245	2.385	2.291	2.312
	PPMD 15	2.276	2.411	2.427	2.399	1.640	2.316	2.214	2.248	2.386	2.291	2.312
	PPME 1	1.867	1.957	1.986	1.950	2.001	1.969	1.999	1.972	1.984	1.880	1.920
	PPME 2	1.847	1.938	1.977	**1.921**	1.940	1.924	1.958	1.938	1.969	**1.872**	1.916
	PPME 3	**1.845**	**1.936**	1.970	1.925	1.937	**1.921**	**1.947**	**1.931**	**1.966**	1.872	**1.911**
	PPME 4	1.847	1.942	**1.964**	1.951	1.946	1.935	1.952	1.941	1.970	1.878	1.912
	PPME 5	1.877	1.966	1.981	2.027	1.979	1.977	1.992	1.985	1.992	1.911	1.932
	PPME 6	1.957	2.046	2.041	2.193	2.047	2.088	2.095	2.095	2.062	1.996	1.989
	PPME 7	2.091	2.217	2.192	2.402	2.084	2.283	2.227	2.250	2.231	2.138	2.118
	PPME 8	2.227	2.403	2.400	2.515	1.952	2.420	2.302	2.333	2.419	2.282	2.285
	PPME 9	2.326	2.523	2.535	2.565	1.743	2.473	2.328	2.367	2.517	2.380	2.408
	PPME 10	2.382	2.577	2.596	2.578	1.624	2.474	2.329	2.372	2.556	2.423	2.464
	PPME 11	2.406	2.592	2.612	2.579	1.581	2.469	2.329	2.373	2.558	2.436	2.481
	PPME 12	2.416	2.595	2.614	2.580	1.572	2.468	2.332	2.375	2.557	2.438	2.483
	PPME 13	2.420	2.595	2.614	2.580	**1.571**	2.467	2.335	2.377	2.556	2.439	2.482
BPPM, $\alpha=0$, $\beta=0.5$	BM-D 1	1.867	1.958	1.986	1.950	2.001	1.969	1.999	1.972	1.984	1.880	1.920
	BM-D 2	1.846	1.938	1.976	**1.919**	1.939	1.923	1.957	1.937	1.968	1.871	1.916
	BM-D 3	1.843	**1.934**	1.969	1.919	1.934	**1.917**	1.943	**1.926**	**1.964**	**1.870**	1.910
	BM-D 4	**1.840**	1.936	**1.960**	1.933	1.933	1.924	**1.939**	1.927	1.965	1.870	**1.907**
	BM-D 5	1.856	1.947	1.967	1.979	1.946	1.948	1.954	1.947	1.975	1.886	1.917
	BM-D 6	1.907	1.996	2.001	2.101	1.973	2.020	2.013	2.017	2.014	1.940	1.949
	BM-D 7	2.025	2.123	2.099	2.325	1.992	2.181	2.152	2.165	2.128	2.065	2.040
	BM-D 8	2.212	2.349	2.314	2.573	1.894	2.408	2.327	2.355	2.350	2.253	2.219
	BM-D 9	2.403	2.588	2.568	2.741	1.736	2.583	2.453	2.498	2.572	2.446	2.439
	BM-D 10	2.551	2.751	2.746	2.821	1.638	2.668	2.522	2.574	2.711	2.590	2.613
	BM-D 11	2.650	2.831	2.836	2.853	1.598	2.703	2.553	2.611	2.774	2.678	2.707
	BM-D 12	2.710	2.865	2.871	2.863	**1.590**	2.718	2.573	2.631	2.799	2.723	2.747
	BM-D 15	2.769	2.883	2.889	2.869	1.618	2.731	2.608	2.663	2.819	2.758	2.767
	BM-D 20	2.777	2.885	2.890	2.871	1.682	2.739	2.642	2.689	2.829	2.764	2.768
	:	:	:	:	:	:	:	:	:	:	:	:

	Method	chmpxx.chr	chntxx.chr	hehcmv.chr	humdyst.chr	humghcs.chr	humhbb.chr	humhdab.chr	humprtb.chr	mpomtcg.chr	mtpacga.chr	vaccg.chr
BPPM, $\alpha=0.5$, $\beta=0.75$	BM-J 1	1.867	1.958	1.986	1.950	2.001	1.970	2.000	1.973	1.984	1.880	1.920
	BM-J 2	1.846	1.938	1.977	1.920	1.939	1.923	1.958	1.937	1.969	1.871	1.916
	BM-J 3	1.843	**1.934**	1.969	**1.919**	1.933	**1.917**	1.943	1.926	**1.964**	1.870	1.910
	BM-J 4	**1.839**	1.935	**1.959**	1.929	1.931	1.922	**1.936**	**1.924**	1.964	**1.868**	**1.906**
	BM-J 5	1.851	1.943	1.964	1.962	1.935	1.938	1.942	1.935	1.971	1.880	1.913
	BM-J 6	1.884	1.977	1.988	2.036	1.935	1.985	1.968	1.971	1.999	1.913	1.934
	BM-J 7	1.945	2.053	2.050	2.136	1.894	2.067	2.008	2.025	2.068	1.975	1.985
	BM-J 8	2.015	2.147	2.150	2.198	1.750	2.134	2.028	2.056	2.159	2.043	2.060
	BM-J 9	2.067	2.208	2.218	2.225	1.586	2.154	2.030	2.066	2.201	2.092	2.118
	BM-J 10	2.100	2.237	2.247	2.238	1.482	2.154	2.032	2.071	2.218	2.118	2.146
	BM-J 15	2.143	2.260	2.271	2.248	1.331	2.145	2.040	2.080	2.222	2.143	2.165
	BM-J 20	2.145	2.260	2.271	2.248	1.306	2.142	2.046	2.085	2.222	2.144	2.164
	BM-J 25	2.145	2.260	2.270	2.248	**1.305**	2.141	2.050	2.088	2.222	2.144	2.164
	BM-J 30	2.145	2.260	2.270	2.248	1.311	2.141	2.052	2.089	2.222	2.144	2.164
	BM-J 40	2.145	2.260	2.270	2.248	1.324	2.141	2.053	2.090	2.223	2.144	2.164
	BM-J 50	2.145	2.260	2.270	2.248	1.334	2.142	2.054	2.090	2.223	2.144	2.164
BPPM, $\alpha=0.5$, $\beta=0.85$	BM-K 1	1.867	1.958	1.986	1.950	2.002	1.970	2.000	1.973	1.984	1.881	1.920
	BM-K 2	1.847	1.938	1.977	1.920	1.939	1.924	1.958	1.937	1.969	1.872	1.916
	BM-K 3	1.843	**1.934**	1.969	**1.919**	1.934	**1.917**	1.943	1.926	**1.964**	1.870	1.910
	BM-K 4	**1.839**	1.935	**1.959**	1.929	1.931	1.922	**1.936**	**1.924**	**1.964**	**1.868**	**1.906**
	BM-K 5	1.850	1.942	1.963	1.960	1.935	1.938	1.941	1.934	1.971	1.879	1.913
	BM-K 6	1.882	1.975	1.987	2.029	1.932	1.981	1.963	1.966	1.997	1.910	1.933
	BM-K 7	1.936	2.046	2.044	2.114	1.886	2.054	1.993	2.010	2.062	1.965	1.979
	BM-K 8	1.993	2.124	2.132	2.154	1.747	2.104	1.995	2.024	2.138	2.019	2.042
	BM-K 9	2.028	2.164	2.178	2.165	1.596	2.107	1.984	2.019	2.159	2.052	2.081
	BM-K 10	2.047	2.176	2.189	2.169	1.504	2.099	1.979	2.016	2.162	2.064	2.091
	BM-K 15	2.068	2.185	2.198	2.174	1.359	2.084	1.979	2.017	2.156	2.072	2.094
	BM-K 20	2.069	2.186	2.197	2.174	1.316	2.078	1.981	2.019	2.153	2.072	2.092
	BM-K 30	2.070	2.186	2.197	2.174	1.293	2.074	1.984	2.021	2.152	2.072	2.092
	BM-K 35	2.070	2.186	2.197	2.174	**1.292**	2.073	1.985	2.021	2.152	2.072	2.092
	BM-K 40	2.070	2.186	2.196	2.174	1.294	2.073	1.985	2.021	2.152	2.072	2.092
	BM-K 50	2.070	2.186	2.196	2.174	1.298	2.073	1.985	2.021	2.152	2.072	2.092
	:	:	:	:	:	:	:	:	:	:	:	:

	Method	chmpxx.chr	chntxx.chr	hehcmv.chr	humdyst.chr	humghcs.chr	humhbb.chr	humhdab.chr	humprtb.chr	mpomtcg.chr	mtpacga.chr	vaccg.chr
BPPM, 7 depth-dep. parameters	DDA 1	1.868	1.959	1.986	1.953	2.003	1.971	2.002	1.975	1.985	1.882	1.920
	DDA 2	1.847	1.938	1.977	1.920	1.939	1.924	1.958	1.938	1.969	1.872	1.916
	DDA 3	1.843	**1.934**	1.969	**1.919**	1.934	**1.918**	1.944	1.927	**1.964**	1.870	1.910
	DDA 4	**1.840**	1.935	**1.959**	1.931	1.933	1.923	**1.938**	**1.926**	1.965	**1.869**	**1.907**
	DDA 5	1.851	1.943	1.964	1.964	1.937	1.940	1.944	1.937	1.972	1.881	1.914
	DDA 6	1.888	1.980	1.990	2.043	1.941	1.989	1.974	1.977	2.001	1.917	1.936
	DDA 7	1.930	2.039	2.040	2.105	1.885	2.047	1.985	2.003	2.056	1.959	1.975
	DDA 8	1.972	2.099	2.109	2.125	1.758	2.077	1.973	2.001	2.113	1.999	2.022
	DDA 9	1.992	2.120	2.135	2.123	1.635	2.069	1.954	1.987	2.118	2.018	2.043
	DDA 10	2.000	2.121	2.134	2.123	1.570	2.059	1.947	1.981	2.112	2.020	2.040
	DDA 15	2.005	2.120	2.133	2.124	1.479	2.046	1.945	1.980	2.103	2.017	2.032
	DDA 20	2.006	2.120	2.132	2.124	1.437	2.041	1.945	1.980	2.100	2.016	2.031
	DDA 30	2.006	2.120	2.132	2.124	1.394	2.034	1.945	1.980	2.098	2.016	2.030
	DDA 40	2.006	2.120	2.131	2.124	1.375	2.031	1.945	1.980	2.097	2.015	2.029
	DDA 50	2.006	2.120	2.131	2.124	**1.367**	2.029	1.945	1.980	2.097	2.015	2.029
BPPM, 6 fanout-dep. parameters	FDB 1	1.868	1.958	1.986	1.951	2.002	1.970	2.000	1.973	1.984	1.881	1.920
	FDB 2	1.847	1.938	1.977	1.921	1.940	1.924	1.959	1.938	1.969	1.872	1.916
	FDB 3	1.843	**1.935**	1.969	**1.920**	1.934	**1.918**	1.944	1.927	1.964	1.870	1.910
	FDB 4	**1.839**	1.935	**1.959**	1.929	1.931	1.922	**1.936**	**1.924**	**1.964**	**1.868**	**1.906**
	FDB 5	1.848	1.940	1.962	1.954	1.931	1.934	1.936	1.930	1.970	1.877	1.911
	FDB 7	1.910	2.017	2.021	2.071	1.863	2.019	1.957	1.974	2.034	1.940	1.957
	FDB 8	1.960	2.079	2.085	2.117	1.726	2.063	1.964	1.990	2.090	1.986	2.004
	FDB 9	1.997	2.124	2.134	2.138	1.570	2.077	1.961	1.994	2.119	2.022	2.042
	FDB 10	2.023	2.147	2.157	2.149	1.466	2.075	1.960	1.996	2.131	2.041	2.062
	FDB 15	2.057	2.166	2.176	2.157	1.304	2.062	1.963	2.002	2.131	2.060	2.076
	FDB 20	2.058	2.166	2.176	2.157	1.272	2.057	1.967	2.005	2.130	2.060	2.075
	FDB 25	2.058	2.166	2.175	2.158	**1.267**	2.056	1.971	2.007	2.130	2.060	2.074
	FDB 30	2.058	2.166	2.175	2.158	1.270	2.056	1.972	2.009	2.130	2.060	2.074
	FDB 40	2.058	2.166	2.175	2.158	1.282	2.056	1.974	2.009	2.130	2.060	2.074
	FDB 50	2.059	2.166	2.175	2.158	1.291	2.057	1.974	2.009	2.131	2.060	2.075
	SM-0	2.778	2.885	2.891	2.871	1.922	2.753	2.677	2.711	2.841	2.769	2.775
	SM-JG	2.071	2.187	2.198	2.176	1.344	2.079	1.989	2.026	2.158	2.074	2.095
	PPMII	1.983	2.084	2.093	2.099	1.286	2.010	1.947	1.985	2.069	2.003	1.996
	ppmz2	**1.903**	**1.992**	**2.013**	**2.017**	**1.263**	**1.922**	**1.890**	**1.912**	**1.980**	**1.923**	**1.909**
	lpaq1	1.819	1.925	1.939	**1.922**	1.202	1.842	1.790	1.830	1.924	1.855	1.852
	paq8l	**1.816**	**1.924**	**1.919**	1.927	**1.201**	**1.839**	**1.783**	**1.821**	**1.923**	**1.851**	**1.847**

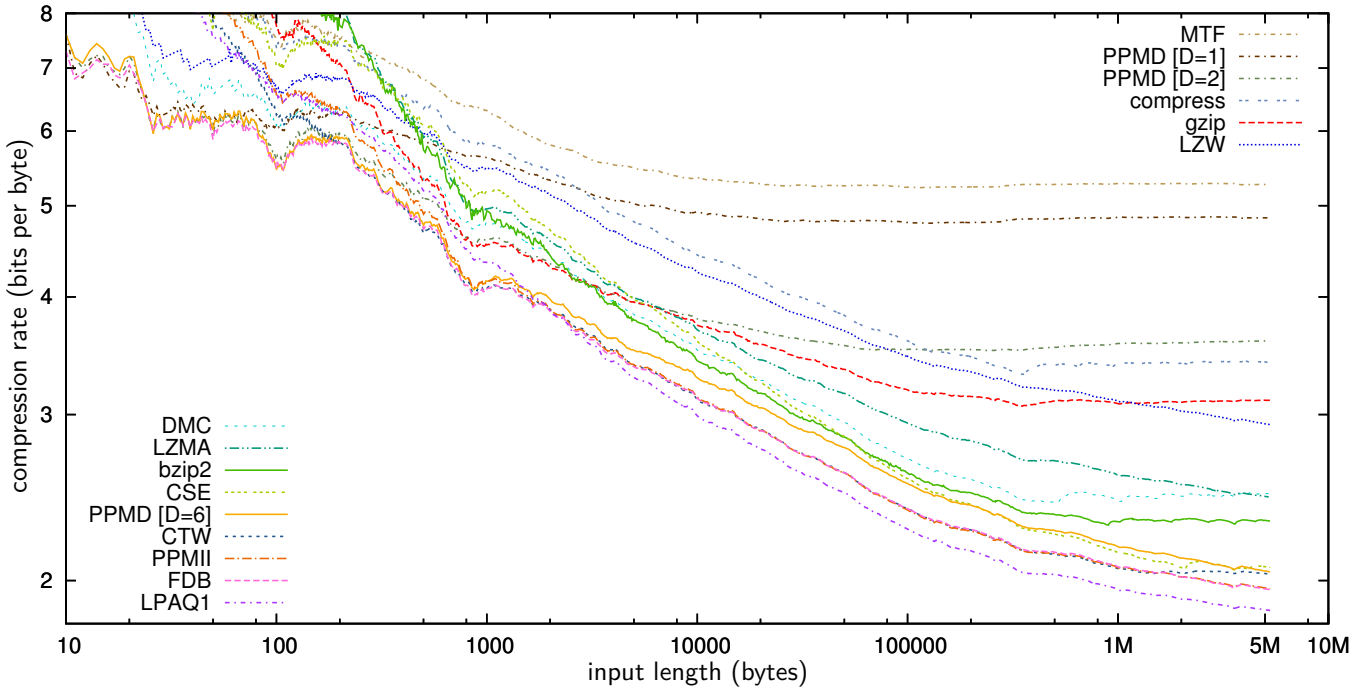


Figure A.1: Compression effectiveness of selected algorithms, in bits per symbol, on Shakespeare's works concatenated in plain text (`shakespeare.txt`). The plot shows how each method's (average) compression effectiveness changes as the file is being processed. For each file position, a truncated version of the input file was made and compressed with each method. The plot shows the length of the compressed output (in bits) divided by the length of the truncated input (in bytes).

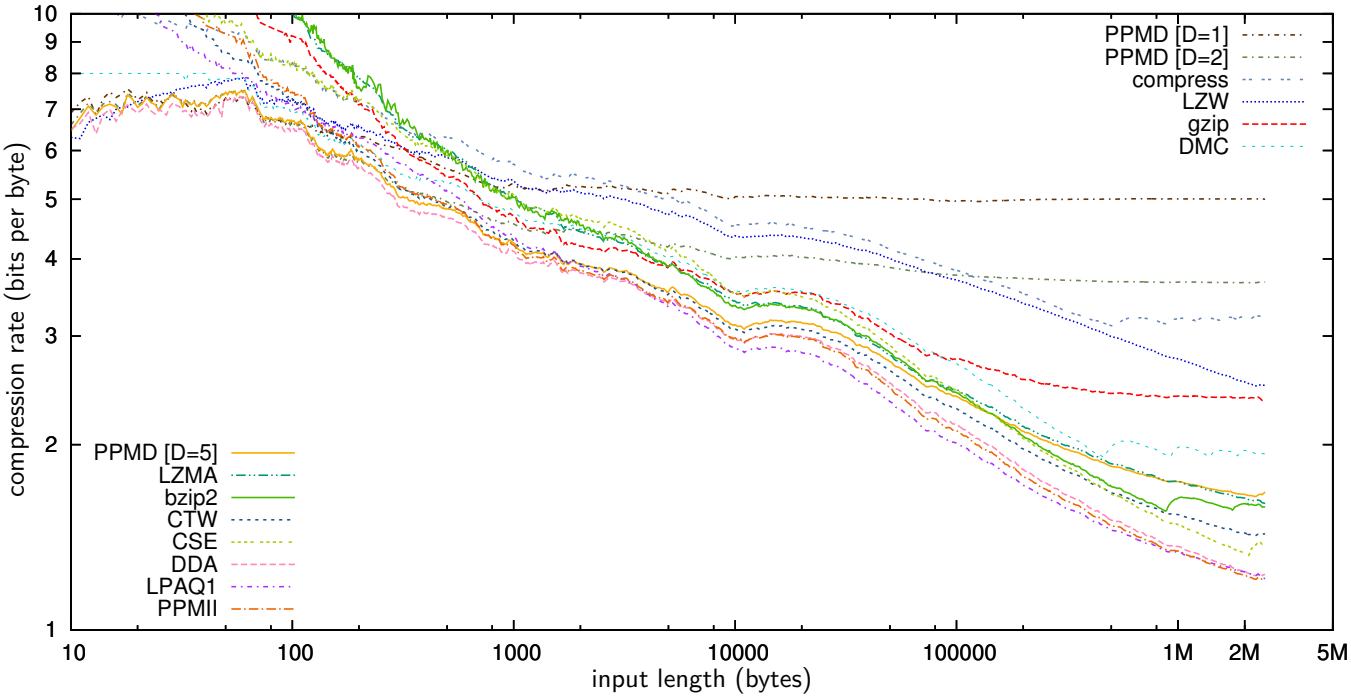
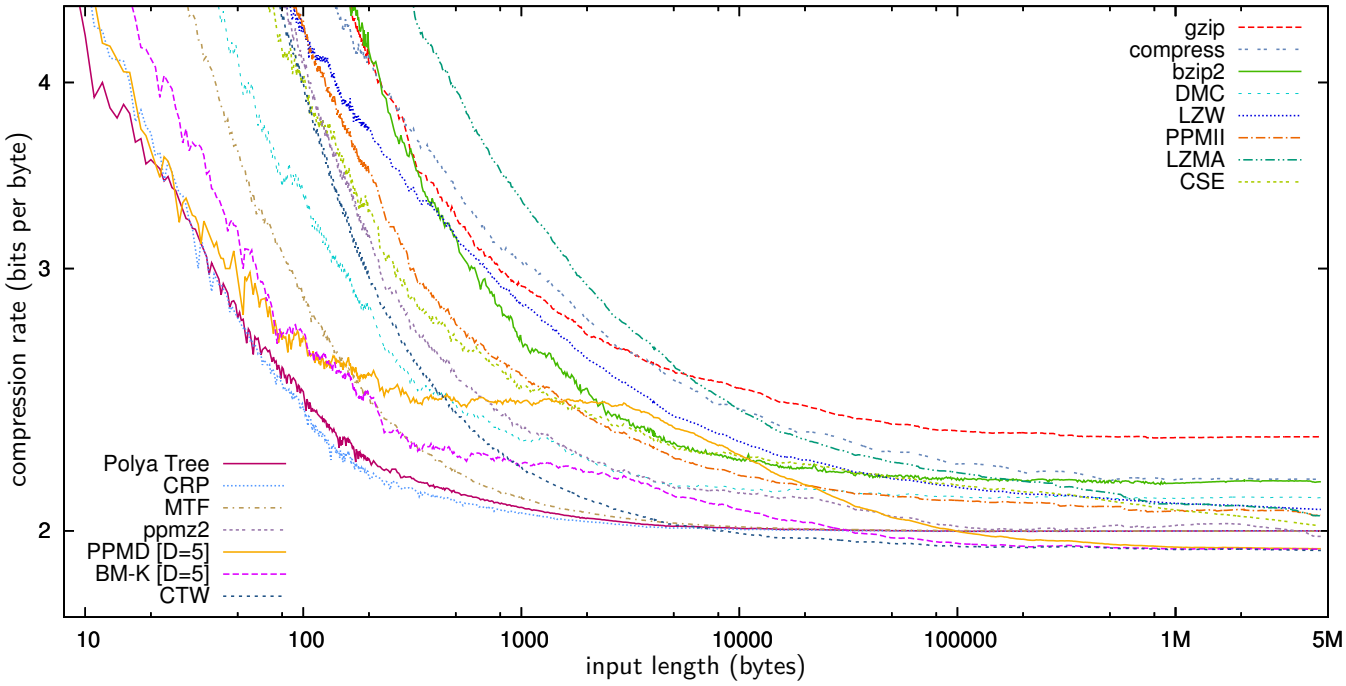
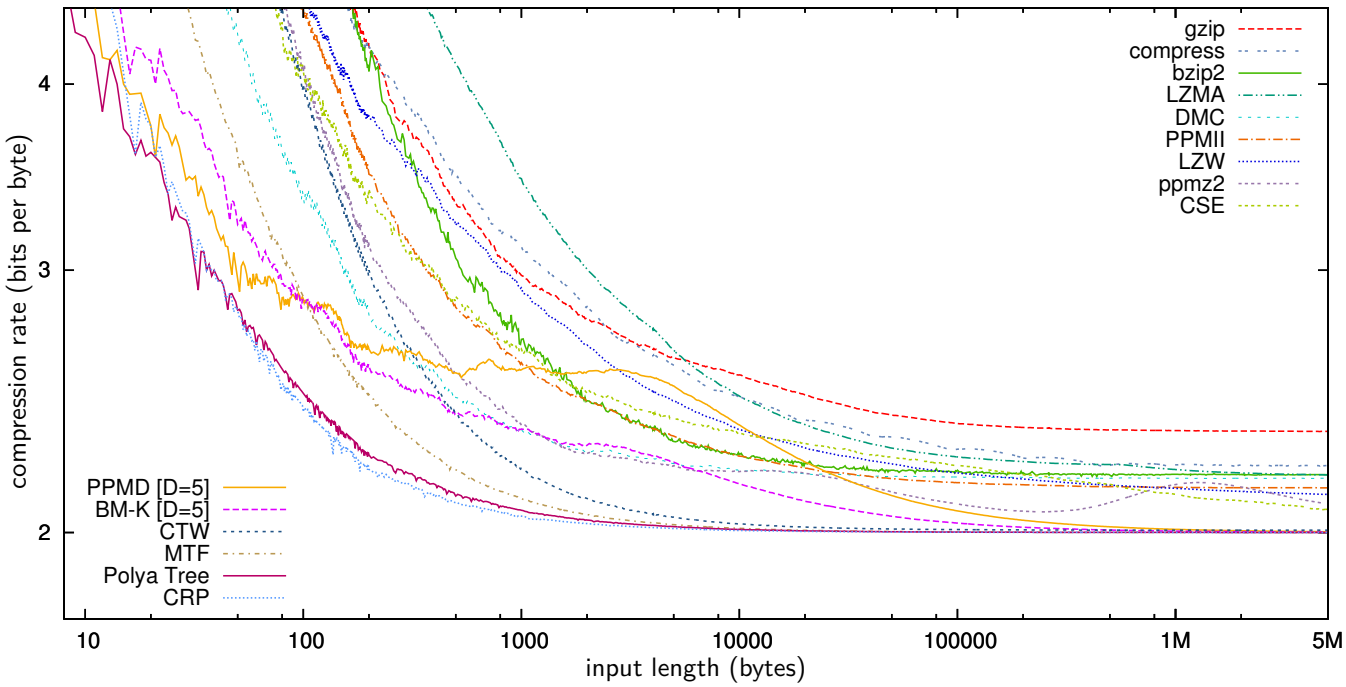


Figure A.2: Progressive compression effectiveness of selected algorithms on `world192.txt`, the CIA world fact book (part of the large Canterbury corpus).



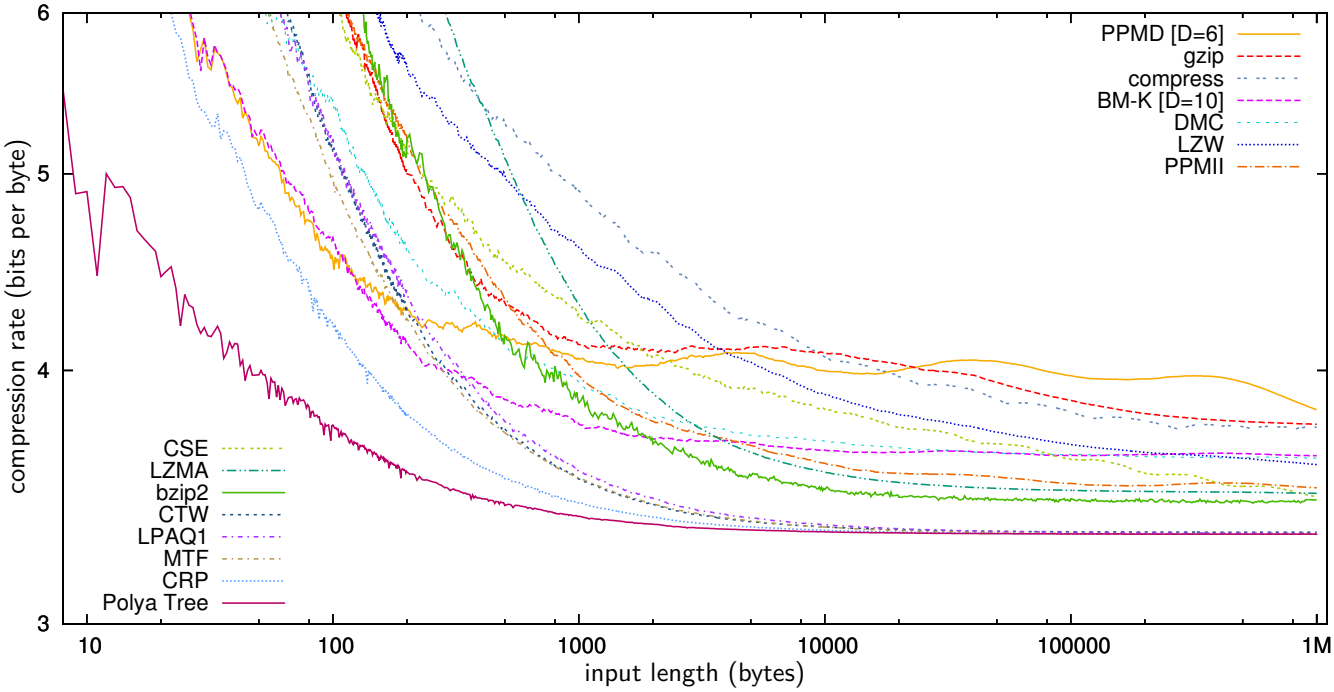
Start: `agcttttcattctgactgcaacgggcaatatgtctctgtgtggattaaaaaaagagtgtctgatagcagcttctgaactgg...`

Figure A.3: Progressive compression effectiveness of selected algorithms on `E.coli`, the complete genome of the bacterium *Escherichia coli*. The sequence is part of the large Canterbury corpus. See Figure A.4 for a comparison with random symbols.



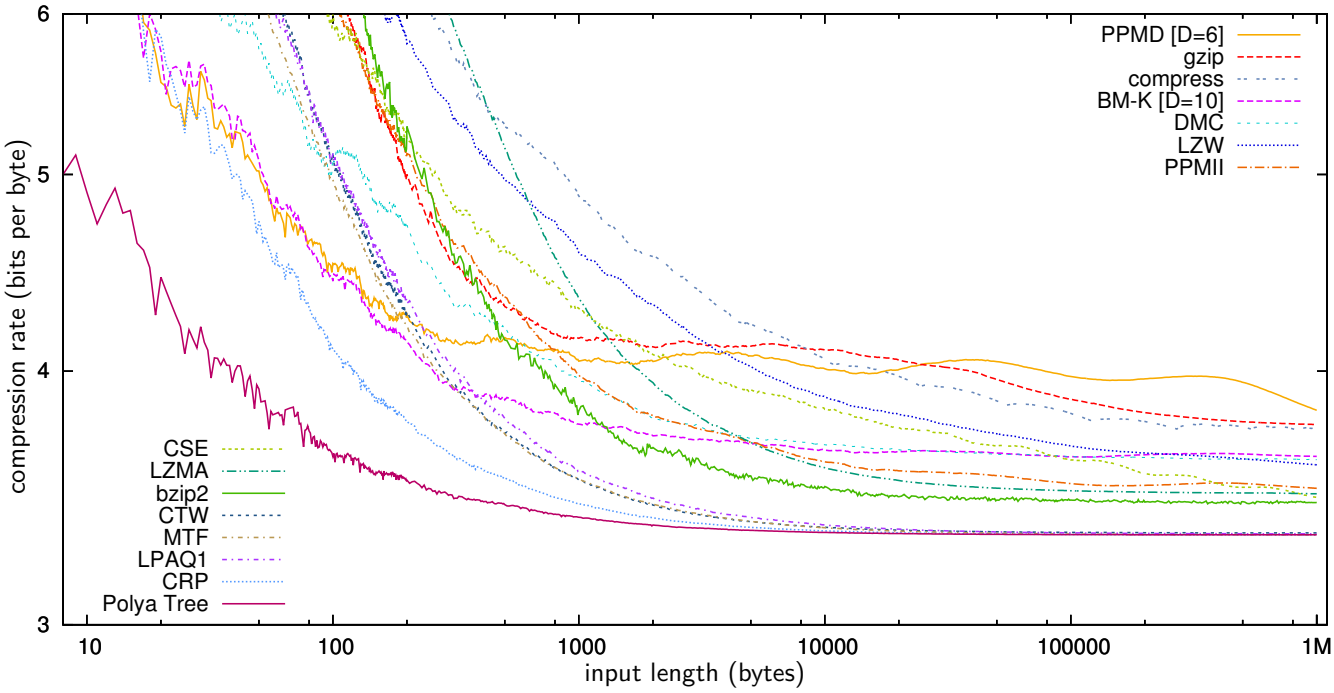
Start: `cacagtgccatggggcgttttaggctgtaacgagtgaacgtccacattaggatagttaccgtgatctacggagggcgcgtg...`

Figure A.4: Progressive compression effectiveness of selected algorithms on a uniformly random sequence (Seq. XVII from chapter 8, page 176). The set of four symbols {a, c, g, t} was chosen to match the nucleotide symbols of the `E.coli` sequence from Figure A.3.



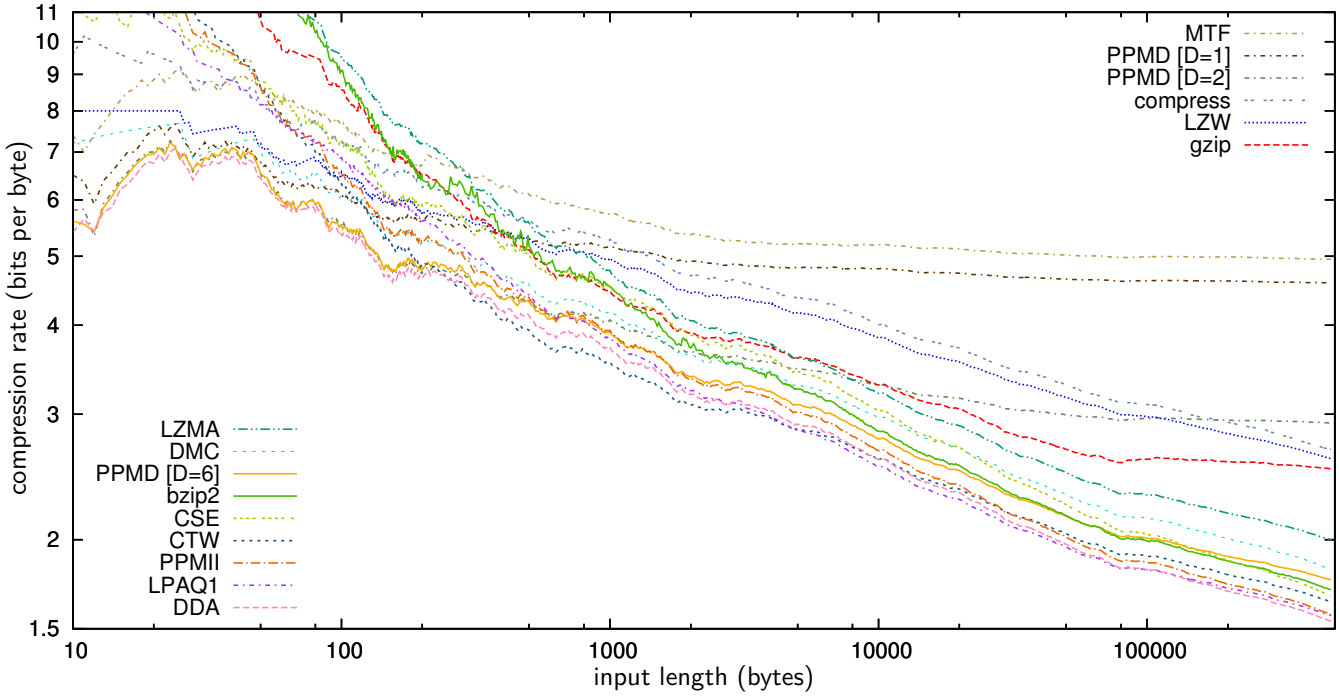
Start: 31415926535897932384626433832795028841971693993751058209749445923078164062862089 9 . . .

Figure A.5: Compression effectiveness of selected algorithms on the decimal expansion of π. This sequence is part of the large Canterbury corpus. See Figure A.6 for a comparison with random symbols.



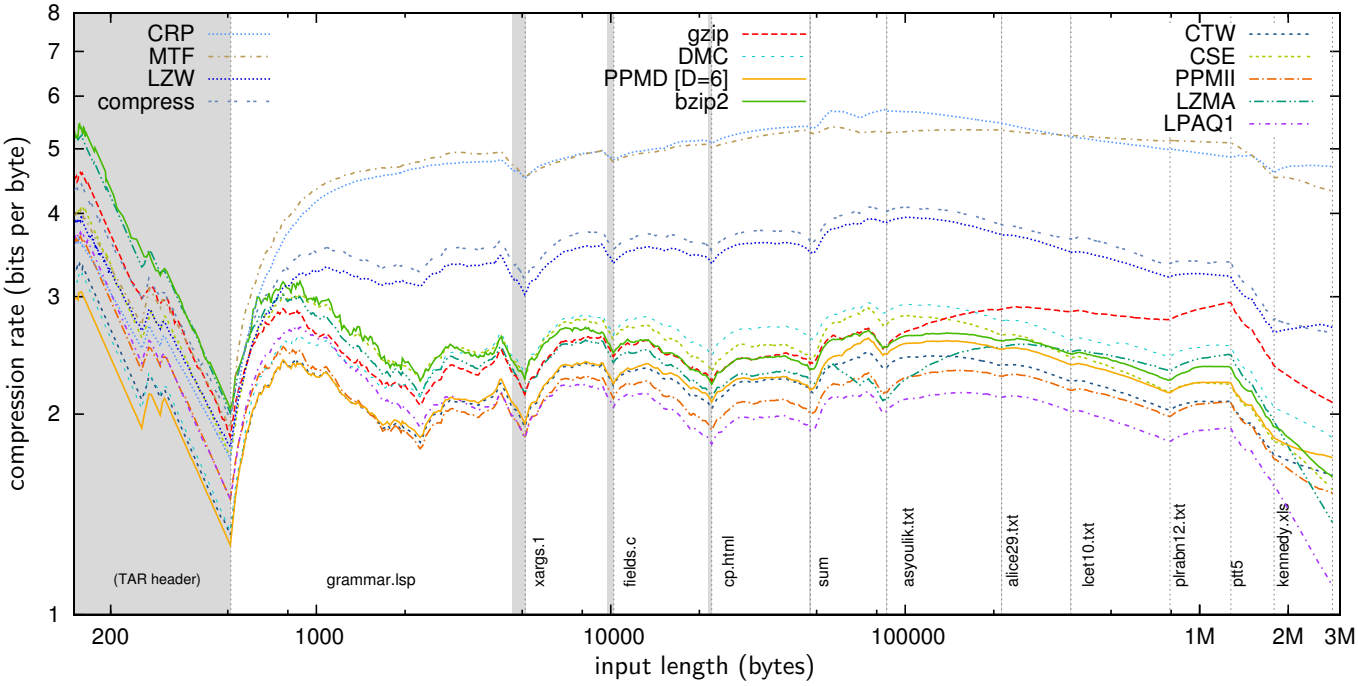
Start: 03840558932262626039036331087606057723349537547573308600391379214489590139336053 0 . . .

Figure A.6: Compression effectiveness of selected algorithms on a sequence of pseudo-random, uniformly distributed decimal digits. See Figure A.5 for a comparison with the decimal expansion of π.



Start: @こころ@@夏目漱石@@@上　先生と私@@一@@　私はその人を常に先生と呼んでいた。だからここ...

Figure A.7: Progressive compression effectiveness of selected algorithms on Japanese novel "Kokoro" by Natsume Sōseki (1914) in UTF–8 encoded plain text.



Start: 63 61 6E 74 65 72 62 75 72 79 2F 67 72 61 6D 6D 61 72 2E 6C 73 70 00 00 00 00 00 00 00 00 00 00 ...

Figure A.8: Compression effectiveness of selected algorithms on a Unix TAR archive of all files in the Canterbury corpus.

Bibliography

Åberg, J. and Shtarkov, Yuri M. (1997). Text compression by context tree weighting. In *Proceedings of the Data Compression Conference*, (edited by James A Storer and Martin Cohn), 377–386. IEEE Computer Society. ISBN 978-0-8186-7761-8. ISSN 1068-0314.

Åberg, Jan, Shtarkov, Yuri M. and Smeets, Ben J. M. (1997). Estimation of escape probabilities for PPM based on universal source coding theory. In *Proceedings of the IEEE International Symposium on Information Theory*. IEEE Computer Society Press. ISBN 978-0-7803-3956-9.

Åberg, Jan, Shtarkov, Yuri M. and Smeets, Ben J. M. (1998). Non-uniform PPM and context tree models. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 279–288. IEEE Computer Society. ISBN 978-0-8186-8406-7. ISSN 1068-0314.

Ahmed, Nasir, Natarajan, T. and Rao, Kamisetty Ramamohan (1974). Discrete cosine transform. *IEEE Transactions on Computers*, **C-23**(1) 90–93. ISSN 0018-9340.

Aldous, David J. (1985). Exchangeability and related topics. *École d'Été de Probabilités de Saint-Flour XIII — 1983*, **1117** 1–198.

ANSI (1986). American national standard for information systems — coded character sets: 7-bit American national standard code for information interchange (7-bit ASCII). American National Standards Institute, ANSI X3.4-1986. Supersedes X3.4-1967 (USAS, 1967).

Apostolico, Alberto and Fraenkel, Aviezri S. (1987). Robust transmission of unbounded strings using Fibonacci representations. *IEEE Transactions on Information Theory*, **33**(2) 238–245. ISSN 0018-9448.

Arnold, Ross and Bell, Tim (1997). A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the Data Compression Conference*, (edited by James A Storer and Martin Cohn). IEEE Computer Society. ISBN 978-0-8186-7761-8. ISSN 1068-0314.

ASA (1963). American Standard Code for Information Interchange. American Standards Association, ASA X3.4-1963. First published version of the ASCII specification, defining binary codes for 100 symbols (Latin uppercase letters, digits, punctuation and control symbols). Superseded by X3.4-1967 (USAS, 1967).

Balkenhol, Bernhard and Kurtz, Stefan (1998). Universal data compression based on the Burrows–Wheeler transformation: theory and practice. Technical Report 98-069, Sonderforschungsbereich: Diskrete Strukturen in der Mathematik, Universität Bielefeld.

Bartlett, Nicholas, Pfau, David and Wood, Frank (2010). Forgetting counts: Constant memory inference for a dependent hierarchical Pitman–Yor process. In *ICML 2010: Proceedings of the 27th International Conference on Machine Learning*, (edited by Johannes Fürnkranz and Thorsten Joachims). Omnipress. ISBN 978-1-60558-907-7.

Bartlett, Nicholas and Wood, Frank (2011). Deplump for streaming data. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Michael W. Marcellin), 363–372. IEEE Computer Society. ISBN 978-1-61284-279-0. ISSN 1068-0314.

BBC (2008). *Dirac Specification, Version 2.2.3*. British Broadcasting Corporation. URL `http://diracvideo.org/`.

Begleiter, Ron and El-Yaniv, Ran (2006). Superior guarantees for sequential prediction and lossless compression via alphabet decomposition. *Journal of Machine Learning Research*, **7** 379–411. ISSN 1532-4435.

Begleiter, Ron, El-Yaniv, Ran and Yona, Golan (2004). On prediction using variable order Markov models. *Journal of Artificial Intelligence Research*, **22**(1) 385–421. ISSN 1076-9757.

Bell, Timothy Clinton, Cleary, John Gerald and Witten, Ian Hugh (1990). *Text Compression*. Prentice Hall. ISBN 978-0-13-911991-0.

Bell, Timothy Clinton, Powell, Matt, Horlor, Joffre and Arnold, Ross (1998). `http://corpus.canterbury.ac.nz/`. Website hosting the files of several compression corpora, including the Canterbury Corpus by Arnold and Bell (1997) and the Calgary Corpus by Bell et al. (1990).

Bender, Edward A. (1974). Partitions of multisets. *Discrete Mathematics*, **9**(4) 301–311. ISSN 0012-365X.

Bentley, Jon Louis, Sleator, Daniel D., Tarjan, Robert E. and Wei, Victor K. (1986). A locally adaptive data compression scheme. *Communications of the ACM*, **29**(4) 320–330. ISSN 0001-0782. The described method is also known as "move-to-front encoding", and was independently discovered by Ryabko (1980).

Berger, Toby, Jelinek, Frederick and Wolf, Jack K. (1972). Permutation codes for sources. *IEEE Transactions on Information Theory*, **18**(1) 160–169. ISSN 0018-9448.

Blackwell, David and MacQueen, James B. (1973). Ferguson distributions via Pólya urn schemes. *The Annals of Statistics*, **1**(2) 353–355. ISSN 0090-5364.

Blasbalg, Herman and van Blerkom, Richard (1962). Message compression. *IRE Transactions on Space Electronics and Telemetry*, **SET-8**(3) 228–238. ISSN 0096-252X.

Bloom, Burton Howard (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7) 422–426. ISSN 0001-0782.

Bloom, Charles (1998). Solving the problems of context modelling. Informally published report. URL `http://cbloom.com/papers/ppmz.pdf`.

Bloom, Charles (2010). Huffman – arithmetic equivalence. Blog post in *cbloom rants*. August 11, 2010. URL `http://cbloomrants.blogspot.co.uk/2010/08/08-11-10-huffman-arithmetic-equivalence.html`.

Bottou, Léon, Howard, Paul Glor and Bengio, Yoshua (1998). The Z-coder adaptive binary coder. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 13–22. IEEE Computer Society. ISBN 978-0-8186-8406-7. ISSN 1068-0314.

Brandenburg, Karlheinz (1999). MP3 and AAC explained. In *Audio Engineering Society Conference: 17th International Conference: High-Quality Audio Coding*. Audio Engineering Society.

de Bruijn, Nicolaas Govert (1946). A combinatorial problem. In *Proceedings of the Section of Sciences*, volume 49, 758–764. Koninklijke Nederlandse Akademie van Wetenschappen.

Bunton, Suzanne (1996). *On-Line Stochastic Processes in Data Compression*. Ph.D. thesis, University of Washington.

Bunton, Suzanne (1997). Semantically motivated improvements for PPM variants. *The Computer Journal*, **40**(2) 76–93. ISSN 0010-4620.

Burrows, Michael and Wheeler, David John (1994). A block-sorting lossless data compression algorithm. Technical Report SRC Research Report 124, Digital Equipment Corporation, Palo Alto, California.

Carroll, Lewis (1865). *Alice's Adventures in Wonderland*. Project Gutenberg. The Millennium Fulcrum Edition 2.9, URL `http://www.gutenberg.org/ebooks/11/`.

Cerf, Vint (1969). ASCII format for network interchange. RFC 20 (Network Working Group) in *Request For Comments*, Internet Engineering Task Force. URL `http://www.ietf.org/rfc/rfc20`.

Champion, Colin J. (1997). A comparison of some techniques for smoothing contingency tables. Technical Report 269, GCHQ (B Division). GHCQ Library and Information Services, PO Box 158, Cheltenham, Gloucestershire GL52 5UG, United Kingdom.

Chen, Stanley F. and Goodman, Joshua (1996). An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, 310–318. Association for Computational Linguistics, Stroudsburg, PA, USA.

Chen, Stanley F. and Goodman, Joshua (1998). An empirical study of smoothing techniques for language modeling. Technical Report TR-10-98, Centre for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, USA.

Cleary, John Gerald and Teahan, William John (1995). Experiments on the zero frequency problem. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 480. IEEE Computer Society. ISBN 978-0-8186-7012-1. ISSN 1068-0314.

Cleary, John Gerald, Teahan, William John and Witten, Ian Hugh (1995). Unbounded length contexts for PPM. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 52–61. IEEE Computer Society. ISBN 978-0-8186-7012-1. ISSN 1068-0314.

Cleary, John Gerald and Witten, Ian Hugh (1984a). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, **32**(4) 396–402. ISSN 0090-6778.

Cleary, John Gerald and Witten, Ian Hugh (1984b). A comparison of enumerative and adaptive codes. *IEEE Transactions on Information Theory*, **30**(2) 306–315. ISSN 0018-9448.

Cormack, Gordon V. (1993). Dynamic Markov compression (`dmc.c`). A file compressor based on the "dynamic Markov compression" algorithm by Cormack and Horspool (1987). Source code. URL `http://plg.uwaterloo.ca/~ftp/dmc/dmc.c`.

Cormack, Gordon V. and Horspool, R. Nigel S. (1987). Data compression using dynamic Markov modelling. *The Computer Journal*, **30**(6) 541–550.

Cover, Thomas M. (1973). Enumerative source encoding. *IEEE Transactions on Information Theory*, **19**(1) 73–77. ISSN 0018-9448.

Cover, Thomas M. and King, Roger C. (1978). A convergent gambling estimate of the entropy of English. *IEEE Transactions on Information Theory*, **24**(4) 413–421. ISSN 0018-9448.

Cover, Thomas M. and Thomas, Joy A. (1991). *Elements of Information Theory*. Wiley, 1st edition. ISBN 978-0-471-06259-2. Superseded by (Cover and Thomas, 2006).

Cover, Thomas M. and Thomas, Joy A. (2006). *Elements of Information Theory*. Wiley, 2nd edition. ISBN 978-0-471-24195-9.

Deutsch, Peter (1996). DEFLATE compressed data format specification version 1.3. RFC 1951 (Informational) in *Request For Comments*, Internet Engineering Task Force. A description of the LZ77 implementation by Katz (1989), URL `http://www.ietf.org/rfc/rfc1951`.

Devroye, Luc (1986). *Non-Uniform Random Variate Generation*. Springer. ISBN 978-1-4613-8643-8. URL `http://cg.scs.carleton.ca/~luc/rnbookindex.html`.

Díaz Díaz, Antonio (2013). `lzip`, version 1.14. A basic command line compressor based on the LZMA SDK by Pavlov (2011). Source code. URL `http://www.nongnu.org/lzip/lzip.html`.

Dubé, Danny and Beaudoin, Vincent (2010). Lossless data compression via substring enumeration. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Michael W. Marcellin), 229–238. IEEE Computer Society. ISBN 978-0-7695-3994-2. ISSN 1068-0314.

Elias, Peter (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, **21**(2) 194–203. ISSN 0018-9448.

Elias, Peter (1987). Interval and recency rank source coding: Two on-line adaptive variable-length schemes. *IEEE Transactions on Information Theory*, **33**(1) 3–10. ISSN 0018-9448.

Ewens, Warren John (1972). The sampling theory of selectively neutral alleles. *Theoretical Population Biology*, **3**(1) 87–112. ISSN 0040-5809.

Fano, Robert Mario (1961). *Transmission of information: a statistical theory of communications*. MIT Press Classics. MIT Press.

Fenwick, Peter M. (1993). A new data structure for cumulative probability tables. Technical Report 88, Department of Computer Science, University of Auckland, New Zealand.

Fenwick, Peter M. (1995). A new data structure for cumulative probability tables: an improved frequency-to-symbol algorithm. Technical Report 110, Department of Computer Science, University of Auckland, New Zealand.

Fenwick, Peter M. (1996). Block sorting text compression — final report. Technical Report 130, Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand.

Fenwick, Peter M. (2007). Burrows–Wheeler compression: Principles and reflections. *Theoretical Computer Science*, **387**(3) 200–219. ISSN 0304-3975.

Ferguson, Thomas S. (1973). A Bayesian analysis of some nonparametric problems. *The Annals of Statistics*, **1**(2) 209–230. ISSN 0090-5364.

Ferguson, Thomas S. (1974). Prior distributions on spaces of probability measures. *The Annals of Statistics*, **2**(4) 615–629. ISSN 0090-5364.

de Finetti, Bruno (1931). Funzione caratteristica di un fenomeno aleatorio. In *Memorie Della Classe Di Scienze Fisiche, Matematiche E Naturali*, volume 4, 251–299. Academia Nazionale dei Lincei.

Fraenkel, Aviezri S. and Klein, Shmuel T. (1996). Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, **64**(1) 31–55.

Frank, Eibe, Chui, Chang and Witten, Ian H. (2000). Text categorization using compression models. Technical Report 00/02, Department of Computer Science, University of Waikato.

Frey, Brendan J. and Hinton, Geoffrey E. (1997). Efficient stochastic source coding and an application to a Bayesian network source model. *The Computer Journal*, **40**(2, 3) 157–165.

Gailly, Jean-loup and Adler, Mark (1992). `gzip`. An open source file compressor based on the DEFLATE algorithm by Katz and Burg (1993). Source code. URL `http://www.gnu.org/software/gzip/`.

Gallager, Robert G. (1978). Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, **24**(6) 668–674. ISSN 0018-9448.

Gasthaus, Jan (2010). `libplump`, version 0.1. A compression library implementing the Deplump algorithm by Gasthaus et al. (2010). Source code. URL `http://www.gatsby.ucl.ac.uk/~ucabjga/code/libplump/libplump-0.1.tar.gz`.

Gasthaus, Jan and Teh, Yee Whye (2010). Improvements to the Sequence Memoizer. In *Advances in Neural Information Processing Systems 23*, (edited by J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel and A. Culotta), 685–693. ISBN 978-1-61782-380-0.

Gasthaus, Jan, Wood, Frank and Teh, Yee Whye (2010). Lossless compression based on the Sequence Memoizer. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Michael W. Marcellin), 337–345. IEEE Computer Society. ISBN 978-0-7695-3994-2. ISSN 1068-0314.

Gerke, Friedrich Clemens (1851). *Der Praktische Telegraphist: oder die Electro-Magnetische Telegraphie: nach dem Morse'schen System, zunächst auch als Handbuch für angehende Telegraphisten*. Hoffmann und Campe, Hamburg. URL `http://books.google.com/books?id=pqlAAAAAcAAJ`.

Ghosh, Jayanta Kumar and Ramamoorthi, R. V. (2002). *Bayesian Nonparametrics*. Springer Series in Statistics. Springer, New York. ISBN 978-0-387-95537-7.

Goblick, T. J., Jr and Holsinger, J. L. (1967). Analog source digitization: A comparison of theory and practice (corresp.). *IEEE Transactions on Information Theory*, **13**(2) 323–326. ISSN 0018-9448.

Golomb, Solomon W. (1966). Run-length encodings. *IEEE Transactions on Information Theory*, **12**(3) 399–401. ISSN 0018-9448.

Gripon, Vincent, Rabbat, Michael, Skachek, Vitaly and Gross, Warren J. (2012). Compressing multisets using tries. In *Information Theory Workshop (ITW)*, 642–646. IEEE. ISBN 978-1-4673-0224-1.

Grumbach, Stéphane and Tahi, Fariza (1993). Compression of DNA sequences. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 340–350. IEEE Computer Society. ISBN 978-0-8186-3392-8. ISSN 1068-0314.

Grumbach, Stéphane and Tahi, Fariza (1994). A new challenge for compression algorithms: genetic sequences. *Information Processing & Management*, **30**(6) 875–886. ISSN 0306-4573.

Highton, Edward (1852). *The electric telegraph: history and progress*. John Weale, 59 High Holborn, London. URL `http://books.google.co.uk/books?id=BRMfAQAAMAAJ`.

Hinton, Geoffrey E. and van Camp, Drew (1993). Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual ACM conference on computational learning theory*, (edited by Lenny Pitt), 5–13. ACM. ISBN 978-0-89791-611-0.

Hinton, Geoffrey E. and Zemel, Richard S. (1994). Autoencoders, minimum description length, and Helmholtz free energy. In *Advances in Neural Information Processing Systems 6*, (edited by Jack D. Cowan, Gerald Tesauro and Joshua Alspector). Morgan Kaufmann. ISBN 978-1-55860-322-6.

Ho, Man-Wai, James, Lancelot F. and Lau, John W. (2006). Coagulation fragmentation laws induced by general coagulations of two-parameter Poisson–Dirichlet processes. Preprint, `http://arxiv.org/abs/math/0601608`, `arXiv:math/0601608`.

Hoppe, Fred M. (1984). Pólya-like urns and the Ewens' sampling formula. *Journal of Mathematical Biology*, **20**(1) 91–94. ISSN 0303-6812.

Howard, Paul Glor (1993). *The Design and Analysis of Efficient Lossless Data Compression Systems*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, Rhode Island 02912, USA.

Howard, Paul Glor and Vitter, Jeffrey Scott (1991). Practical implementations of arithmetic coding. Technical Report CS-91-45, Department of Computer Science, Brown University, Providence, Rhode Island, USA. A revised version exists, see Howard and Vitter (1992).

Howard, Paul Glor and Vitter, Jeffrey Scott (1992). Practical implementations of arithmetic coding. Technical Report CS-92-18, Department of Computer Science, Brown University, Providence, Rhode Island, USA. Revised version of an earlier report (Howard and Vitter, 1991).

Huffman, David Albert (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, **40**(9) 1098–1101. ISSN 0096-8390.

Hutter, Marcus (2001). Towards a universal theory of artificial intelligence based on algorithmic probability and sequential decisions. In *Machine Learning: ECML 2001*, (edited by Luc de Raedt and Peter Flach), volume 2167, 226–238. Springer Berlin Heidelberg.

Hutter, Marcus (2004). *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin. ISBN 978-3-540-22139-5. URL `http://www.hutter1.net/ai/uaibook.htm`.

Hutter, Marcus (2006). Prize for compressing human knowledge. A data compression contest measured on a 100 MB extract of Wikipedia. URL `http://prize.hutter1.net/`.

Ishwaran, Hemant and James, Lancelot F. (2001). Gibbs sampling methods for stick-breaking priors. *Journal of the American Statistical Association*, **96**(453) 161–173. ISSN 0162-1459.

ISO (1993). *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio*. International Standard ISO/IEC 11172–3:1993, JTC1/SC29 WG11. International Organization for Standardization, Geneva, Switzerland.

ISO (1994). *Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines*. International Standard ISO/IEC 10918–1:1994, JTC1/SC29 WG1. International Organization for Standardization, Geneva, Switzerland.

ITU-R (2004). *Recommendation M.1677: International Morse Code*. International Telecommunication Union, Radiocommunication Sector (ITU-R). Superseded by (ITU-R, 2009).

ITU-R (2009). *Recommendation M.1677-1: International Morse Code*. International Telecommunication Union, Radiocommunication Sector (ITU-R). Supersedes (ITU-R, 2004).

ITU-T (1992). *Recommendation T.81: Information technology – Digital compression and coding of continuous-tone still images – Requirements and guidelines*. International Telecommunication Union, Telecommunications Standardization Sector (ITU-T).

ITU-T (1993). *Recommendation H.82: Progressive Bi-Level Image Compression*. International Telecommunication Union, Telecommunications Standardization Sector (ITU-T).

ITU-T (2003). *Recommendation H.264: Advanced video coding for generic audiovisual services*. International Telecommunication Union, Telecommunications Standardization Sector (ITU-T).

JPEG (1992). *The JPEG standard*. Joint Photographic Experts Group. Standardised by ITU-T (1992) and ISO (1994).

Katz, Phil (1989). `APPNOTE.TXT` — .ZIP file format specification. Distributed by PKWARE Inc. (Milwaukee, Wisconsin, USA). First published version. For the latest version, see (Peterson et al., 2012).

Katz, Phil (1993). `APPNOTE.TXT` — .ZIP file format specification, version 2.0. Distributed by PKWARE Inc. (Milwaukee, Wisconsin, USA). Contains the first published description

of the DEFLATE algorithm developed by Katz and Burg (1993). For the latest version of `APPNOTE.TXT`, see (Peterson et al., 2012).

Katz, Phil and Burg, Steve (1993). `PKZIP`, version 2.04g. Software. Distributed by PKWARE Inc. (Milwaukee, Wisconsin, USA). First implementation of the DEFLATE algorithm, described by Katz (1993).

Kautz, William H. (1965). Fibonacci codes for synchronization control. *IEEE Transactions on Information Theory*, **11**(2) 284–292. ISSN 0018-9448.

Kingman, John Frank Charles (1993). *Poisson Processes*. Oxford University Press, Great Clarendon Street, Oxford, OX2 6DP, England. ISBN 978-0-19-853693-2.

Kneser, Reinhard and Ney, Hermann (1995). . In *International Conference on Acoustics, Speech, and Signal Processing*, volume 1, 181–184. ISSN 1520-6149.

Knoll, Byron and de Freitas, Nando (2012). A machine learning perspective on predictive coding with PAQ8. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Michael W. Marcellin), 377–386. IEEE Computer Society. ISBN 978-1-4673-0715-4. ISSN 1068-0314.

Knuth, Donald Ervin (1985). Dynamic Huffman coding. *Journal of Algorithms*, **6**(2) 163–180. ISSN 0196-6774. An improvement to the algorithm by Gallager (1978).

Knuth, Donald Ervin (1998). *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison Wesley Longman, 2nd edition. ISBN 978-0-201-89685-5.

Knuth, Donald Ervin (2004). Generating all permutations. Fascicle 2B in *The Art of Computer Programming, Volume IV* (pre-print).

Knuth, Donald Ervin (2005a). Generating all combinations. Fascicle 3A in *The Art of Computer Programming, Volume IV* (pre-print).

Knuth, Donald Ervin (2005b). Generating all partitions. Fascicle 3B in *The Art of Computer Programming, Volume IV* (pre-print).

Kolmogorov, Andrey Nikolaevich (1963). On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, **25**(4) 369–376. ISSN 0972-7671.

Kolmogorov, Andrey Nikolaevich (1965). Три подхода к определению понятия «количество информации» (Three approaches to the definition of the concept "quantity of information"). Проблемы Передачи Информации *(Problems of Information Transmission)*, **1**(1) 3–11. ISSN 0555-2923. In Russian.

Kolmogorov, Andrey Nikolaevich (1968). Logical basis for information theory and probability theory. *IEEE Transactions on Information Theory*, **14**(5) 662–664. ISSN 0018-9448.

Korodi, Gergely and Tabus, Ioan (2008). On improving the PPM algorithm. In *3rd International Symposium on Communications, Control and Signal Processing*, 1450–1453. ISBN 978-1-4244-1687-5.

Kullback, Solomon and Leibler, Richard A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, **22**(1) 79–86. ISSN 0003-4851.

Langdon, Glen G., Jr (1984). Arithmetic coding. *IBM Journal of Research and Development*, **28**(2) 149–162. ISSN 0018-8646.

Laplace, Pierre-Simon (1814). *Essai philosophique sur les probabilités*. Mme. Ve. Courcier, Imprimeur-Libraire pour les Mathématiques, Quai des Augustins N^{o}57, Paris.

Lavine, Michael (1992). Some aspects of Polya tree distributions for statistical modelling. *The Annals of Statistics*, **20**(3) 1222–1235. ISSN 0090-5364.

Lehmer, D. H. (1958). Teaching combinatorial tricks to a computer. In *Combinatorial analysis: Proceedings of the Tenth Symposium in Applied Mathematics of the American Mathematical Society*, (edited by Richard Ernest Bellman and Marshall Hall), volume 10. AMS.

Levenshtein, Vladimir I. (1968). Об избыточности и замедлении разделимого кодирования натуральных чисел. Проблемы кибернетики, **20** 173–179. In Russian. ("On the redundancy and delay of decodable coding of natural numbers." *Problemy kibernetiki*.).

MacKay, David J. C. (2002). `macopt` — a nippy wee optimizer. Source code and documentation. URL `http://www.inference.phy.cam.ac.uk/mackay/c/macopt.html`.

MacKay, David J. C. (2003). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press. ISBN 978-0-521-64298-9. URL `http://www.inference.phy.cam.ac.uk/mackay/itila/`.

MacKay, David J. C. and Bauman Peto, Linda Charlene (1995). A hierarchical Dirichlet language model. *Natural Language Engineering*, **1**(3) 289–307. ISSN 1351-3249.

Mahoney, Matthew Vincent (1999). Text compression as a test for artificial intelligence. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999)*, (edited by Jim Hendler and Devika Subramanian), 970. AAAI Press. ISBN 978-0-262-51106-3.

Mahoney, Matthew Vincent (2000). Fast text compression with neural networks. In *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2000)*, (edited by James N. Etheredge and Bill Z. Manaris), 230–234. AAAI Press. ISBN 978-1-57735-113-9.

Mahoney, Matthew Vincent (2002). The PAQ1 data compression program. Unpublished draft, URL `http://cs.fit.edu/~mmahoney/compression/paq1.pdf`.

Mahoney, Matthew Vincent (2005). Adaptive weighing of context models for lossless data compression. Technical Report CS-2005-16, Department of Computer Science, Florida Institute of Technology, Melbourne, FL, USA.

Mahoney, Matthew Vincent (2007a). `lpaq1` – a leightweight version of PAQ. Source code. URL `http://cs.fit.edu/~mmahoney/compression/lpaq1.zip`.

Mahoney, Matthew Vincent (2007b). `paq8l` open source file compressor and archiver. Source code. URL `http://cs.fit.edu/~mmahoney/compression/paq8l.zip`.

Mauldin, R. Daniel, Sudderth, William D. and Williams, S. C. (1992). Polya trees and random distributions. *The Annals of Statistics*, **20**(3) 1203–1221. ISSN 0090-5364.

Mayzner, Mark S. and Tresselt, Margaret Elizabeth (1965). Tables of single-letter and digram frequency counts for various word-length and letter-position combinations. *Psychonomic Monograph Supplements*.

Michel, Jean-Baptiste, Shen, Yuan Kui, Presser Aiden, Aviva, Veres, Adrian, Gray, Matthew K., the Google Books team, Pickett, Joseph P., Hoiberg, Dale, Clancy, Dan, Norvig, Peter, Orwant, Jon, Pinker, Steven, Nowak, Martin A. and Lieberman Aiden, Erez (2011). Quantitative analysis of culture using millions of digitized books. *Science*, **331**(6014) 176–182. ISSN 0036-8075.

Mnih, Andriy and Hinton, Geoffrey E. (2009). A scalable hierarchical distributed language model. In *Advances in Neural Information Processing Systems 21*, (edited by Daphne Koller, Dale Schuurmans, Yoshua Bengio and Léon Bottou), 1081–1088. Curran Associates, Inc. ISBN 978-1-60560-949-2.

Moffat, Alistair (1990). Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, **38**(11) 1917–1921. ISSN 0090-6778.

Moffat, Alistair (1999). An improved data structure for cumulative probability tables. *Software: Practice and Experience*, **29**(7) 647–659. ISSN 1097-024X.

Moffat, Alistair and Katajainen, Jyrki (1995). In-place calculation of minimum-redundancy codes. In *Algorithms and Data Structures*, (edited by Selim G. Akl, Frank Dehne, Jörg-Rüdiger Sack and Nicola Santoro), volume 955 of *Lecture Notes in Computer Science*, 393–402. Springer Berlin Heidelberg. ISBN 978-3-540-60220-0.

Moffat, Alistair, Neal, Radford M. and Witten, Ian H. (1998). Arithmetic coding revisited. *ACM Transactions on Information Systems*, **16**(3) 256–294. ISSN 1046-8188.

Morse, Samuel Finley Breese (1840). Improvement in the mode of communicating information by signals by the application of electro-magnetism. US Patent 1647.

MPEG (1991). *Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio*. Moving Picture Experts Group. Committee Draft, standardised by ISO (1993). Origin of the MPEG–1 Layer III (MP3) audio encoding.

Nádas, Arthur (1984). Estimation of probabilities in the language model of the IBM speech recognition system. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **32**(4) 859–861. ISSN 0096-3518.

Natsume, Sōseki〔夏目漱石〕(1914). *Kokoro*〔こころ〕. From Aozora Bunko library〔青空文庫〕— converted from ruby annotated HTML to UTF–8 plain text, URL `http://www.aozora.gr.jp/cards/000148/card773.html`.

NIST (1995). Secure hash standard. Federal Information Processing Standards (FIPS), Publication 180-1. Information Technology Laboratory, National Institute of Science and Technology (NIST), USA. Origin of the SHA-1 algorithm. For a more recent version, see (NIST, 2012).

NIST (2012). Secure hash standard. Federal Information Processing Standards (FIPS), Publication 180-4. Information Technology Laboratory, National Institute of Science and Technology (NIST), USA.

Norvig, Peter (2013). English letter frequency counts: Mayzner revisited. Modern version of the letter frequency analysis by Mayzner and Tresselt (1965), based on data from the Google books N-gram corpus (Michel et al., 2011), URL `http://norvig.com/mayzner.html`.

Öktem, Levent (1999). *Hierarchical Enumerative Coding and Its Applications in Image Compression*. Ph.D. thesis, Signal Processing Laboratory, Tampere University of Technology.

O'Neal, J. B., Jr (1967). A bound on signal-to-quantizing noise ratios for digital encoding systems. *Proceedings of the IEEE*, **55**(3) 287–292. ISSN 0018-9219.

O'Neal, J. B., Jr (1971). Entropy coding in speech and television differential PCM systems (corresp.). *IEEE Transactions on Information Theory*, **17**(6) 758–761. ISSN 0018-9448.

Pasco, Richard Clark (1976). *Source coding algorithms for fast data compression*. Ph.D. thesis, Department of Electrical Engineering, Stanford University.

Pavlov, Igor (2003). 7-Zip, version 3.13. Source Code, first published version. URL `http://www.7-zip.org/`.

Pavlov, Igor (2009). 7-Zip, version 9.04 beta. Source Code. URL `http://www.7-zip.org/`.

Pavlov, Igor (2011). LZMA SDK. Source Code. URL `http://www.7-zip.org/sdk.html`.

Pennebaker, William B., Mitchell, Joan L., Langdon, Glen G., Jr and Arps, Ronald B. (1988). An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, **32**(6) 717–726. ISSN 0018-8646.

Perman, Mihael (1990). *Random discrete distributions derived from subordinators*. Ph.D. thesis, Department of Statistics, University of California, Berkeley.

Perman, Mihael, Pitman, Jim and Yor, Marc (1992). Size-biased sampling of Poisson point processes and excursions. *Probability Theory and Related Fields*, **92**(1) 21–39. ISSN 0178-8051.

Peterson, Jim et al. (2006). `APPNOTE.TXT` — .ZIP file format specification, version 6.3.0. Distributed by PKWARE Inc. (Milwaukee, Wisconsin, USA). URL `http://www.pkware.com/documents/APPNOTE/APPNOTE-6.3.0.TXT`.

Peterson, Jim et al. (2012). `APPNOTE.TXT` — .ZIP file format specification, version 6.3.3. Distributed by PKWARE Inc. (Milwaukee, Wisconsin, USA). URL `http://www.pkware.com/documents/APPNOTE/APPNOTE-6.3.3.TXT`.

Pitman, Jim (1999). Coalescents with multiple collisions. *The Annals of Probability*, **27**(4) 1870–1902. ISSN 0091-1798.

Pitman, Jim and Yor, Marc (1995). The two-parameter Poisson–Dirichlet distribution derived from a stable subordinator. Technical Report 433, Department of Statistics, University of California, 367 Evans Hall #3860, Berkeley, CA 94720-3860.

Propp, James G. and Wilson, David B. (1996). Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random structures and Algorithms*, **9**(1-2) 223–252. ISSN 1098-2418.

Propp, James G. and Wilson, David B. (1997). Coupling from the past: a user's guide. In *Microsurveys in Discrete Probability*, (edited by David Aldous and James Propp), volume 41 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 181–192. AMS. ISBN 978-0-8218-0827-6.

Reznik, Yuriy A. (2011). Coding of sets of words. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Michael W. Marcellin), 43–52. IEEE Computer Society. ISBN 978-1-61284-279-0. ISSN 1068-0314.

Rissanen, Jorma J. (1976). Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, **20**(3) 198–203. ISSN 0018-8646.

Rissanen, Jorma J. and Langdon, Glen G., Jr (1979). Arithmetic coding. *IBM Journal of Research and Development*, **23**(2) 149–162. ISSN 0018-8646.

Rissanen, Jorma J. and Langdon, Glen G., Jr (1981). Universal modeling and coding. *IEEE Transactions on Information Theory*, **27**(1) 12–23. ISSN 0018-9448.

Ryabko, Boris Yakovlevich (1980). Сжатие данных с помощью стопки книг (Data compression by means of a "book stack"). Проблемы Передачи Информации *(Problems of Information Transmission)*, **16**(4) 16–19. ISSN 0555-2923. Also known as "move-to-front encoding", independently discovered by Bentley et al. (1986).

Sadakane, Kunihiko, Okazaki, Takumi and Imai, Hiroshi (2000). Implementing the context tree weighting method for text compression. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 123–132. IEEE Computer Society. ISBN 978-0-7695-0592-3. ISSN 1068-0314.

Sayir, Jossy (1999). *On Coding by Probability Transformation*. Ph.D. thesis, Swiss Federal Institute of Technology Zurich.

Seward, Julian (1997). `bzip2`, version 0.1. A compressor based on the block-sorting transform of Burrows and Wheeler (1994). Source code, first published version.

Seward, Julian (2010). `bzip2`, version 1.0.6. A compressor based on the block-sorting transform of Burrows and Wheeler (1994). Source code. URL `http://www.bzip.org/`.

Shannon, Claude Elwood (1948). A mathematical theory of communication. *The Bell System Technical Journal*, **27**(3) 379–423, 623–656. ISSN 0005-8580.

Shannon, Claude Elwood (1949). Communication theory of secrecy systems. *The Bell System Technical Journal*, **28**(4) 656–715. ISSN 0005-8580.

Shannon, Claude Elwood (1950). Prediction and entropy of printed English. *The Bell System Technical Journal*, **30**(1) 50–64. ISSN 0005-8580.

Shkarin, Dmitry A. (2001a). Повышение эффективности алгоритма PPM (Improving the efficiency of the PPM algorithm). Проблемы Передачи Информации *(Problems of Information Transmission)*, **37**(3) 44–54. ISSN 0555-2923. For an English translation see Shkarin (2001b).

Shkarin, Dmitry A. (2001b). Improving the efficiency of the PPM algorithm. *Problems of Information Transmission*, **37**(3) 226–235. ISSN 1608-3253. Translated from Russian (Shkarin, 2001a).

Shkarin, Dmitry A. (2002). PPM: One step to practicality. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 202–211. IEEE Computer Society. ISBN 978-0-7695-1477-2. ISSN 1068-0314.

Shkarin, Dmitry A. (2006). `ppmdj`. An implementation of the PPMII algorithm by Shkarin (2001a). Source code. URL `http://www.compression.ru/ds/ppmdj.rar`.

Smith, Fred W. (1967). U.S.A. standard code for information interchange. *Western Union Technical Review*, 184–191. Review article about X3.4-1967 (USAS, 1967).

Solomonoff, Ray J. (1964a). A formal theory of inductive inference. Part I. *Information and Control*, **7**(1) 1–22. ISSN 0019-9958.

Solomonoff, Ray J. (1964b). A formal theory of inductive inference. Part II. *Information and Control*, **7**(2) 224–254. ISSN 0019-9958.

Stanley, Richard Peter (1986). *Enumerative Combinatorics – Volume 1.* Wadsworth & Brooks/Cole Mathematics Series. Wadsworth Publishing Company, Belmont, CA, USA. ISBN 978-0-412-98261-3.

Steinruecken, Christian (2014). Compressing sets and multisets of sequences. In *Proceedings of the Data Compression Conference*, (edited by Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà and James A. Storer), 427. IEEE Computer Society. ISBN 978-1-4799-3882-7. ISSN 1068-0314.

Storer, James A. and Szymanski, Thomas G. (1982). Data compression via textual substitution. *Journal of the ACM*, **29**(4) 928–951. ISSN 0004-5411.

Teahan, William John and Harper, David John (2001). Using compression-based language models for text categorization. In *Proceedings of the Workshop on Language Modeling and Information Retrieval*, (edited by Jamie Callan, Bruce Croft and John Lafferty), 83–87. Pittsburgh, Pennsylvania, USA.

Teahan, William John, Inglis, Stuart J., Cleary, John Gerald and Holmes, Geoffrey (1998). Correcting English text using PPM models. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn). IEEE Computer Society. ISBN 978-0-8186-8406-7. ISSN 1068-0314.

Teh, Yee Whye (2006a). A Bayesian interpretation of interpolated Kneser–Ney. Technical Report TRA2/06, School of Computing, National University of Singapore.

Teh, Yee Whye (2006b). A hierarchical Bayesian language model based on Pitman–Yor processes. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, 985–992.

Teh, Yee Whye (2010). Dirichlet process. In *Encyclopedia of Machine Learning*, (edited by Claude Sammut and Geoffrey I. Webb). Springer. ISBN 978-0-387-30768-8.

Teh, Yee Whye, Jordan, Michael I., Beal, Matthew J. and Blei, David M. (2006). Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, **101**(476) 1566–1581. ISSN 0162-1459.

Teuhola, Jukka (1978). A compression method for clustered bit-vectors. *Information Processing Letters*, **7** 308–311. ISSN 0020-0190.

Thomas, Spencer W., McKie, Jim, Davies, Steve, Turkowski, Ken, Woods, James A. and Orost, Joe (1985). `compress.c`, revision 4.0. A file compressor based on the LZW algorithm by Welch (1984). Source code.

Turing, Alan Mathison (1950). Computing machinery and intelligence. *Mind*, **LIX**(236) 433–460. ISSN 0026-4423.

Ukkonen, Esko (1995). On-line construction of suffix trees. *Algorithmica*, **14**(3) 249–260. ISSN 0178-4617.

Unicode (1991). The Unicode Standard, version 1.0. The Unicode Consortium (Mountain View, CA, USA). First published version of the Unicode Standard. For latest version, see Unicode (latest).

Unicode (latest). The Unicode Standard. The Unicode Consortium (Mountain View, CA, USA). URL `http://www.unicode.org/versions/latest/`.

USAS (1967). USA Standard Code for Information Interchange. United States of America Standards Institute, USAS X3.4-1967. Extends X3.4-1963 (ASA, 1963) by adding lowercase letters, circumflex and underscore, and tidying up the control characters. The original document is unavailable, see (Smith, 1967) for a summary of the contents. Superseded by X3.4-1986 (ANSI, 1986).

Vail, Alfred (1845). *The American Electro Magnetic Telegraph: With the Reports of Congress, and a Description of all Telegraphs Known, Employing Electricity or Galvanism*. Philadelphia: Lea & Blanchard. URL `http://google.co.uk/books?id=jeYOAAAAYAAJ`.

Varshney, Lav R. and Goyal, Vivek K. (2006a). Ordered and disordered source coding. In *Proceedings of the Information Theory & Applications Inaugural Workshop*.

Varshney, Lav R. and Goyal, Vivek K. (2006b). Toward a source coding theory for sets. In *Proceedings of the Data Compression Conference*, (edited by James A. Storer and Martin Cohn), 13–22. IEEE Computer Society. ISBN 978-0-7695-2545-7. ISSN 1068-0314.

Varshney, Lav R. and Goyal, Vivek K. (2007). On universal coding of unordered data. In *Information Theory and Applications Workshop 2007, Conference Proceedings*, 183–187. IEEE.

Vitter, Jeffrey Scott (1987). Design and analysis of dynamic Huffman codes. *Journal of the ACM*, **34**(4) 825–845. ISSN 0004-5411.

Walker, Stephen G., Damien, Paul, Laud, Purushottam W. and Smith, Adrian F. M. (1999). Bayesian nonparametric inference for random distributions and related functions. *Journal of the Royal Statistical Society, Series B (Statistical Methodology)*, **61**(3) 485–527. ISSN 1369-7412.

Ward, David J., Blackwell, Alan F. and MacKay, David J. C. (2000). Dasher — a data entry interface using continuous gestures and language models. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, (edited by Mark S. Ackerman and Keith Edwards), UIST 2000, 129–137. ACM. ISBN 978-1-58113-212-0.

Welch, Terry A. (1984). A technique for high-performance data compression. *IEEE Computer*, **17**(6) 8–16. ISSN 0018-9162.

Wiegand, Thomas and Schwarz, Heiko (2011). Source coding: Part I of fundamentals of source and video coding. *Foundations and Trends in Signal Processing*, **4**(1) 1–222. ISSN 1932-8346.

Willems, Frans M. J. (1989). Universal data compression and repetition times. *IEEE Transactions on Information Theory*, **35**(1) 54–58. ISSN 0018-9448.

Willems, Frans M. J. (1998). The context-tree weighting method: Extensions. *IEEE Transactions on Information Theory*, **44**(2) 792–798. ISSN 0018-9448.

Willems, Frans M. J., Shtarkov, Yuri M. and Tjalkens, Tjalling J. (1993). Context tree weighting: A sequential universal source coding procedure for FSMX sources. In *International Symposium on Information Theory, Proceedings*, 59. IEEE. ISBN 978-0-7803-0878-7.

Willems, Frans M. J., Shtarkov, Yuri M. and Tjalkens, Tjalling J. (1995). The context-tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, **41**(3) 753–664. ISSN 0018-9448.

Witten, Ian Hugh and Bell, Timothy Clinton (1991). The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, **37**(4) 1085–1094. ISSN 0018-9448.

Witten, Ian Hugh, Neal, Radford and Cleary, John Gerald (1987). Arithmetic coding for data compression. *Communications of the ACM*, **30**(6) 520–540. ISSN 0001-0782.

Wood, Frank (2011). Modeling streaming data in the absence of sufficiency. Presented at "Bayesian Nonparametrics: Hope of Hype?", workshop at NIPS 2011.

Wood, Frank, Archambeau, Cédric, Gasthaus, Jan, James, Lancelot and Teh, Yee Whye (2009). A stochastic memoizer for sequence data. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, (edited by Léon Bottou and Michael L. Littman), volume 382 of *ACM International Conference Proceeding Series*, 1129–1136. ISBN 978-1-60558-516-1.

Wood, Frank, Gasthaus, Jan, Archambeau, Cédric, James, Lancelot and Teh, Yee Whye (2011). The Sequence Memoizer. *Communications of the ACM*, **52**(2) 91–98. ISSN 0001-0782.

Zaks, Shmuel (1980). Lexicographic generation of ordered trees. *Theoretical Computer Science*, **10**(1) 63–82. ISSN 0304-3975.

Zeckendorf, Édouard (1972). Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas. *Bulletin de la Société Royale des Sciences de Liège*, **41**(3–4) 179–182. ISSN 0037-9565.

Ziv, Jacob and Lempel, Abraham (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, **IT-23**(3) 337–343. ISSN 0018-9448.

Ziv, Jacob and Lempel, Abraham (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, **IT-24**(5) 530–536. ISSN 0018-9448.

Index

Entry types
Text: N, Graphs: $N^{\bigstar}$, Tables: N^{T},
Algorithms: N^{A}, Bibliography: N^{B}

*not dissimilar to this one

"What an age of wonders is this! When one considers the state of Science a century ago, and compares the light of the past with that of the present day—how great is the change! how marvellous the advance! Discovery has followed discovery in rapid succession—invention has superseded invention—till it would seem to the superficial observer that little now remains to be discovered, and that further improvement is next to an impossibility."

—— Edward Highton (1852)
'The Electric Telegraph: Introduction'

EOF