On-Line Stochastic Processes in Data Compression

by

Suzanne Bunton

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by_____
(Chairperson of Supervisory Committee)

_____

_____

Program Authorized
to Offer Degree_____

Date_____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

On-Line Stochastic Processes in Data Compression

by Suzanne Bunton

Chairpersons of Supervisory Committee:
Professors Gaetano Borriello and Richard Ladner
Department of Computer Science and Engineering

The ability to predict the future based upon the past in finite-alphabet sequences has many applications, including communications, data security, pattern recognition, and natural language processing. By Shannon's theory and the breakthrough development of arithmetic coding, any sequence, $a_1 a_2 \cdots a_n$, can be encoded in a number of bits that is essentially equal to the minimal information-lossless codelength, $\sum_i -\log_2 p(a_i|a_1 \cdots a_{i-1})$. The goal of universal on-line modeling, and therefore of universal data compression, is to deduce the model of the input sequence $a_1 a_2 \cdots a_n$ that can estimate each $p(a_i|a_1 \cdots a_{i-1})$ knowing only $a_1 a_2 \cdots a_{i-1}$ so that the expected value of $-\log p(a_i|a_1 \cdots a_{i-1})$ is minimized. Thus, *data compression* has become both a routine application of on-line modeling techniques and a means for accurately measuring their empirical performance.

The on-line modeling algorithm, Prediction By Partial Matching (PPM), has set the performance standard in data compression research since its introduction in 1984. PPM's success stems from its *ad hoc* probability estimator, which dynamically blends distinct frequency distributions contained in a single model into a probability estimate for each input symbol. Meanwhile, the most conclusive asymptotic results use an information-theoretic metric to dynamically select a model from a set of competing models, and then use that selected model to estimate the currently scanned symbol's probability. Our hypothesis is that these apparently unrelated approaches can be combined to produce a semantically coherent technique that is arguably universal and which consistently outperforms existing techniques on actual data.

To prove our hypothesis, we first give a semantics that unifies both forms of on-line modeling. Then we show how related but linguistically distinct model families fit the semantics, and give a new frequency update mechanism that is consistent with the semantics for all families. Next, we generalize PPM's probability estimator to a family of estimators, which we combine with a novel model-selection mechanism that eliminates the need for the order bounds and suboptimal hill-climbing employed by previous techniques. We organize into a cross-product the set of estimators, the set of known model selectors (including our mechanism), and the set of linguistic structures used by three model families. The result is an executable taxonomy, which we use for evaluating these techniques in experiments that control *all* model features, as well as test data.

The combination of our update mechanism, generalization of blending, and novel model selector outperforms the best known published approaches by 5% to 12% (measured as compression performance on the Calgary corpus), the largest published increase since 1990. In addition to delivering superior performance, we reduce memory requirements and increase the fundamental understanding of the problem of universal sequence prediction and of the relationships among the previously known on-line solutions.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

I wish to express my sincerest appreciation of the five people who directly influenced the quality of this work and made its completion possible. Gaetano Borriello provided years of steady support and freedom to pursue my own interests—even though they eventually diverged from his own. Richard Ladner provided his keen error-detection abilities, and listened critically and tirelessly to my ideas during the last year and a half of this work. Dick Karp's thoughtful editing clarified my baroque prose and poorly defined mathematical constructions. Preston Briggs edited many drafts, provided countless hours of technical assistance with LaTeX and Unix, and served as audience for repeated practice talks. Lastly, I am deeply indebted to Tim Hunkapiller, my current employer, for allocating funds to the completion of this work.

The following friends and colleagues provided technical assistance over the years: Ross Williams acted as a critical sounding board for my early ideas; and Bruno Carpentieri, Ian Witten, Tim Bell, and Alistair Moffat carefully listened to an *ad hoc* presentation of an early version of this thesis at DCC93. Victor Miller and Ian Witten 'sanity-checked' my opinionated first chapters. Glen Langdon provided encouragement and some very useful references to variants of the algorithm *Context*. John Cleary enthusiastically accepted my ideas when we first met at DCC95. When I presented the existing state-selectors and their drawbacks to him, he suggested the basic dynamic programming approach that I used in this thesis to eliminate those drawbacks. Jim Storer and Marty Cohn funded my attendance at DCC93, and then at DCC95 with the Renato Capocelli Prize for that year's submission. Thank heavens for my closest buddies: Victor Miller, Preston Briggs, Wendy Thrash, Angela Thalls, Neil McKenzie, Jim Wiggs, and Melanie Lewis Fulgam. As the only student or faculty in our department who worked in my research area, I had to rely

upon my own evaluation of the significance of my ideas while they were in development. Because of the resulting isolation, and the extensive nature of my project, completing this work required, above all else, years of sustained faith. These loyal, deeply introspective, and candid friends helped me regain my perspective during repeated periods of self-doubt.

And while the topic is emotional support, I want to add that the warmth, competence, and sense of humor of our department's graduate student coordinator, Frankye Jones, has punctuated my stay here with fond memories. Bless you, Frankye!

Chapter 1

INTRODUCTION

All data must be encoded before they can be manipulated, stored, or transmitted by a computer. Encoding schemes can serve various purposes: cryptographic schemes provide a measure of security; error-detecting or error-correcting schemes enable recovery from transmission errors; and data compression schemes minimize code lengths so that storage and communication resources can be better utilized. Usually, however, computer data are encoded simply, for convenient access.

Textual computer data provide a classic example of coding for convenient access. Textual data are encoded in 8-bit bytes, which are viewed as characters from a 256-symbol alphabet. Similarly, almost all other computer data are represented as fixed-size symbols from much larger alphabets, where each symbol consists of a fixed number of bytes. It is the *fixed size* of these symbol codes, combined with their statistically non-uniform appearances in actual data sequences, that provide the opportunity for compression. Minimum code lengths are possible only when frequent symbols are mapped to short codes and infrequent symbols are mapped to necessarily longer codes. All data compression techniques therefore perform, at least implicitly, probability estimation.

The performance of a data compressor is measured by average code length—the length of the compressed sequence divided by the size of the original sequence. Current research that seeks to improve upon the compression performance of the best-performing techniques known relies upon explicit, sophisticated data models for probability estimation. Furthermore, the problem of finding a means for minimally describing a sequence is extremely well-motivated [Ris89], and has application to other fields such as pattern recognition, language processing, and cryptography.

However, there is a limit to the compressibility of any sequence, and new techniques necessarily work harder than their predecessors to effect ever-decreasing improvements in compression. From a computer systems perspective, data compres-

sion is merely an optimization. System designers certainly desire increased effective storage and communication bandwidth, but compression and decompression must consume fewer or less valuable resources than the resources saved. Thus, data compression is a field divided. Theoreticians strive for provably minimal expected code length, while practitioners gladly sacrifice some code-length minimality for substantial savings in computational resources.

1.1 Why On-Line Stochastic Data Compression?

The last decade saw the gap separating CPU speed and memory capacity from secondary storage capacity and access latency widen considerably. As a consequence, on-line statistical data compression techniques, which are relatively computationally intensive, have risen from "laboratory curiosities" into practice [WMB94, Chapter 2]. Members of this family of techniques have out-performed almost every other type of compressor for the past thirteen years.[1] Moreover, the information-theoretical underpinnings of statistical techniques enable rigorous analysis. One goal of this work is to improve upon the known theoretical and practical data compression techniques that use general, explicit probability estimation. Below, I justify the confinement of my discussion to *stochastic* techniques for *on-line, lossless* data compression.

1.1.1 Lossless vs. Lossy Data Compression

Lossless data compression is the reversible re-encoding of a data sequence so that the length of the re-encoded sequence is smaller than the length of the original. Lossless compression is appropriate for computing applications that require the decompressed data to be identical to the original data. In contrast, *lossy* data compression techniques are applicable in situations where the decoded data can still be useful if they are noisy. Speech, image, and video data are three popular applications of lossy coding techniques, and these are all examples of *signal* data.

Signal data consist of finitely represented samples of analog signals. To treat these finitely represented quantities as members of an alphabet, as we do with *computer* data, such as text or object code, is to grossly overgeneralize the problem, and thereby

[1] One notable exception is the off-line lossless transform method published by Burroughs and Wheeler [BW94], which is competitive with the best on-line stochastic techniques.

throw away useful *a priori* knowledge. Adjacent samples in a signal data sequence are usually highly correlated, whereas adjacent symbols in alphabet-oriented computer data are not. In cases where the signal data are very expensive to gather, it may be worthwhile to apply lossless coding techniques, but the algorithms should be tailored to exploit the characteristics of signal data, rather than alphabet-oriented computer data.

Lossy compression techniques oriented towards signal data are one to two orders of magnitude more effective at reducing the average code length of signal-data sequences than are lossless techniques oriented towards alphabet-data sequences. The correlations within the signal sequences explain only part of the difference. Lossy methods have the additional freedom to throw away part of the original information to effect the reduction. Even so, the distorted data that result must still be encoded, preferably with short codes assigned to probable events and long codes assigned to improbable events. Furthermore, the best lossy techniques restrict the amount of information lost with probable samples more than with improbable samples.[2] In other words, lossy techniques strive to minimize the expected per-sample distortion (with respect to an appropriate distortion measure), as well as the expected per-sample code length. Because computing the probability of each (decorrelated[3]) sample and of each sample's distorted value is required to implement both optimizations, the data modeling techniques for probability estimation explored in this thesis are doubly applicable to the lossy compression of signal data.

1.1.2 On-Line vs. Multiple-Pass Data Compression

On-line data compression techniques are *adaptive*[4] and *universal*; that is, they only require a single pass over the data sequence and employ no prior knowledge about the sequence, other than the source alphabet. All data compressors use some sort

---

[2] This is a topic of rate distortion theory, which is rigorously presented in [CT91], and applied to tree-structured pattern classification in [Cho88]. An application to image compression, pruned tree-structured vector quantization [Cho88, Chapter 5], is further developed in [Ris90].

[3] Discrete signal data sequences can easily be decorrelated, and thereafter treated as alphabet sequences suitable for modeling with alphabet-oriented techniques, by first recoding each successive sample as the difference between it and the preceding sample.

[4] The information-theoretic literature employs the term *sequential*.

of data model, either explicitly or implicitly, and on-line compressors incrementally build a model as they encode the data sequence. The model used for coding later in a given source sequence will (hopefully) be more representative than the model used earlier in the sequence.

In most modeling applications, off-line modeling is an alternative to on-line modeling. With off-line modeling, the model is built in a preprocessing pass, and the model is said to be *trained off-line*. A description of an off-line model must be transmitted in addition to the encoded source message: the shortest such description is known as the Minimal Description Length (MDL) [Ris89] of the sequence.

On-line models, which are incrementally reconstructed by the decoder from the decoded data sequence, require no specific transmission. However, the prediction inaccuracies endured early in the source stream by on-line techniques increase the overall length of the encoded sequence, just as a static-model preamble would. It has been shown [BCW90, CW84a, Ris89] that we can approach the MDL with on-line models, and that there exist circumstances where an off-line model will perform arbitrarily worse than any on-line model. In practice, on-line models do work better than off-line models. Furthermore, with on-line models we need not conceive of a coding scheme for efficiently transmitting the model itself, and we only require one pass over the message sequence.

1.1.3 Stochastic vs. Ziv-Lempel Compression

The best approaches to on-line adaptive data compression, in terms of compression performance, combine an on-line adaptive stochastic process with an arithmetic coder. At each time step i, the stochastic process takes as input the event a_i from a sequence $a_1 a_2 \cdots a_n$ of instances of symbols from a finite alphabet, and outputs a probability estimate $P_e(a_i)$. The arithmetic coder takes as input the $P_e(a_i)$ and codes the symbol instance a_i in $-\log_2 P_e(a_i)$ bits. If the stochastic process used to model the input sequence is identical to the machine which emitted the sequence, the number of bits output by the combined model and arithmetic coder will approach the entropy of the source message, which is the theoretical optimum [Sha48].

The relative generality and power of stochastic techniques justify the exclusion of data compression methods that do not separate the problem into modeling for probability estimation and coding. Though only a few practical data compression meth-

ods use explicit stochastic models, researchers have systematically proved that the other known practical methods can be exactly and efficiently emulated with adaptive stochastic models [Ris83, Lan83, Bel86]. Furthermore, the most effective practical data compression method for the past thirteen years, namely PPM, is a stochastic technique [Bel86, BWC89, BCW90, CTW95, CW84b, Mof90, WMB94].

Other techniques, namely any variant of the elegant Ziv-Lempel techniques [ZL77, ZL78, MW85, Wel84, FG89, BB92], trade compression performance for speed and memory conservation more appropriately for today's technology. These techniques are examples of the more general textual substitution techniques [Sto85]. Textual substitution techniques are much easier than stochastic techniques to understand and implement, and therefore save another costly resource—design time.[5]

1.1.4 The Importance of Stochastic Techniques

It is undeniable that some of the stochastic methods we discuss, in their current state of development, are only minimally practical. Their generality and optimality are costly. Today's stochastic models greedily consume resources in terms of design time, integration cost (such as timing and buffering overhead in hardware implementations), and of course, execution speed and memory requirements. However, there are several ways that the in-depth study of these techniques can prove valuable.

First there is the epistemological goal of better understanding the abstract problem of modeling sequences on line. Perhaps the solution to the problem may turn out to be easy to approximate in an implementation that is very practical and only imperceptibly less effective than the full computation.[6] Also, since CPU cycles are becoming cheaper and shorter at a faster rate than storage technology is improving, the additional compression gain of these computationally intensive methods could become inexpensive enough to be worthwhile.

On the other hand, the overwhelming result of this collective research effort may

---

[5] Suffice it to say that if this author needed to design—within the next few months—an implementation that fit on a single chip, had good compression performance, required trivial timing and buffering overhead, and was an order of magnitude faster than the state of the art, she would abandon the techniques presented here, just as she did seven years ago [BB92, Bun92].

[6] Indeed, this has already happened in the case of arithmetic coding [RM89, CKW91, HV92, FGC93, MSWB93].

be that it is simply not worthwhile to improve the compression performance further. Even so, there remains one other very important motivation for this research. These same stochastic modeling techniques are applicable to other computing disciplines in a straightforward manner. Any absolute improvement in the accuracy of probability estimates only affects codelength by an amount equal to its base 2 logarithm. For example, in this thesis, we introduce an algorithm that gets an average codelenth of 2.177 bits per character (bpc) on the files of the Calgary Corpus. The benchmark PPM implementation [Mof90], gets 2.48 bpc on the same data. These codelengths correspond to expected probability estimates of 0.1792 and 0.2211 per character, respectively. Therefore, our algorithm improves upon PPM's compression performance by 12.2%, while improving the probability estimates by 23.4%. Because of the lesser effect that improved probability estimates have on codelengths, the additional predictive ability of our models will make more difference in applications that use probability estimates directly to aid in decision making.

Thus, on-line stochastic modeling techniques arguably provide the most powerful and general lossless data compression techniques known, and it is likely that their additional power will be worth the additional resource costs in the future. In the following chapters, we explore the best stochastic techniques known within the data compression, communications, and information-theoretic literature.

1.2 Thesis Overview

This document addresses the problem of deducing a finite-state source model, belonging to a tractable FSM subclass, from a discrete-time, finite-alphabet source message, *on-line*. The default application of solutions to this problem is lossless source coding for communication channels, that is, data compression, where the goal is to increase effective channel bandwidth. However, there are many applications of the modeling process itself, including natural language understanding and speech recognition. All of these applications share a common goal: to construct a model from a training sequence that will assign the highest possible probabilities to members of the set of sequences represented by the training data. Thus the application of modeling techniques developed in this thesis to other domains is straightforward.

1.2.1 A Brief Tour of Coding and Modeling Concepts

Although the data streams encountered in practice are rarely emitted by probabilistic automata, the assumption that they are makes the modeling problem tractable, and lends insight to the problem of modeling data streams. Thus the modeling goal is to deduce a close approximation of the probabilistic automaton that presumably emitted the source message, from the source message itself. To further simplify the problem, we assume that source messages were generated by restricted finite state machines (FSMs), such as finite-order FSMs.

The general approach to modeling with finite-order FSMs is to gather *a posteriori* observations from the source message and to record these observations as frequency counts. Frequency data are organized into counters for symbols at individual FSM states. Each state corresponds to particular sets of subsequences over the input alphabet. The sets are called conditioning contexts and they form a partition over A^*, where A is the finite input alphabet. Generally speaking, a state's counter for a given symbol a is incremented whenever an a is seen in the input sequence immediately following a subsequence that is a member of the state's conditioning context. For example, consider the conditioning context consisting of strings ending in abc. Whenever $abcb$ appears in the input stream, the frequency of b given the context A^*abc is incremented. The states of the probabilistic FSMs correspond one-to-one to members of the conditioning context partition, and the FSM model structure determines the context partition to a large degree. (In Chapter 5 we show that the details of how frequency updates are performed can be another defining factor of a model's context partition.)

The two key subproblems of deducing a finite-order FSM information source from a given input sequence (or source message) are

1. computing the best partition of conditioning contexts for a given source message, and
2. estimating probabilities of source message symbols from the observations as they are organized within a given context partition.

The context partition may be computed off-line in a separate pass over the source message, or on-line, in which case the symbol probabilities are estimated from the symbol frequencies organized in the partition formed from the already-processed portion of the message. In the on-line case, probability estimation from *a posteriori* observations

is complicated by the so-called *zero-frequency problem*, that is, the problem of assigning a non-zero probability to a novel event. Also, the on-line problem of incrementally computing the model structure requires dynamic *local order estimation*, where *order* is the minimal length of the substrings that distinguish the conditioning context of a particular state. Local order estimation is used to detect where the model needs refining, or alternatively, to detect where the model needs pruning. Dynamic model pruning is equivalent to selecting, from among a set of competing models, a model suitable for estimating the probability of the currently scanned symbol.

Classes of models are defined by the general structure of their conditioning-context partitions. One well-known, tractable class of information sources, which is properly contained in the family of finite-order FSMs, is the FSMX class. The FSMX class contains many of the well-known practical on-line models in the literature, including PPM (Prediction by Partial Matching), which has been the best-performing technique for 13 years. The class also contains some well-known theoretical constructions used for proving asymptotic results, including Rissanen's 'Context' algorithm, and a recent algorithm (which we shall denote WLZ) proved by Weinberger, Lempel, and Ziv to be asymptotically optimal for the FSMX class.

1.2.2 Thesis Statement

My thesis, or principal claim, is that the local-order estimation techniques that dominate information-theoretic approaches can be combined with the solutions to the zero-frequency problem that dominate practical approaches, for enhanced performance. Theoretical on-line algorithms perform local order estimation by simulating several competing partitions of conditioning contexts simultaneously. For each source symbol, the model corresponding to the best-performing partition is selected, and then used to estimate the symbol's probability. The goal in theoretical analysis is to construct a model that converges on the assumed source asymptotically, so these techniques typically neglect the subtleties of estimating probabilities from *finite* source messages. In contrast, practical on-line algorithms simulate a single partition of conditioning contexts, which is periodically refined using heuristics such as global order bounds that impose arbitrary assumptions that severely restrict the class of assumed source models. However, the best practical models, exemplified by PPM, solve the zero-frequency problem so effectively that symbol probabilities are accu-

rately estimated in the face of scarce message statistics. The result is that purely information-theoretic techniques are designed to perform well on long sequences while practical techniques are designed to perform well on short sequences. A meaningful combination of these approaches should yield superior performance for sequences of any length.

1.2.3 Thesis Contributions

My approach to proving the above claim is to break down known theoretical and practical algorithms into solutions to the same general set of component subproblems:

- the type of model structure (Chapters 3 and 4),
- the organization and updating of frequency data (Chapter 5),
- how the coding model is selected from the simulated models (Chapter 7), and
- how the coding distribution is estimated using the coding model (Chapter 6).

The decomposition of the algorithms is based upon a unifying semantics, which is developed in primers throughout the thesis (Sections 2.6.3, 5.1, 7.4, and 7.6). The primers make explicit the many assumptions about the input sequence inherent to each component solution. In order to decompose the Dynamic Markov Compression (DMC) algorithm, I formally identified and proved its (previously unknown) structural semantics in Chapter 4. The corollary that DMC's Markov model is not FSMX disproves the widely held belief that the FSMX class contains the class of Markov models. Additionally, the component decomposition and unifying semantics make obvious better solutions to most component subproblems.

The component subproblems and their solutions are described in Chapters 5–7. The cross product of the sets of solutions to each of these components defines a taxonomy of on-line suffix-tree modeling algorithms for sequences, which I describe in Chapter 8. The taxonomy succinctly identifies the mathematical relationships among several independently developed techniques, and is fully implementable. The resulting *executable taxonomy* re-implements several known algorithms, implements improved versions of those algorithms, and implements novel combinations of their components.

In Chapter 9, I use the executable taxonomy for conducting controlled experiments. Experiments with the Calgary Corpus conclusively support my hypothesis

that the combination of information-theoretic model selection and PPM's probability estimator is superior to either group of techniques alone. I perform controlled comparative studies of all solutions to the problems of coding model selection and coding distribution estimation. The studies demonstrate conclusively that my improvements to PPM's model construction, DMC's model construction, PPM's coding distribution estimator, and a new technique for coding model selection, are each superior to their predecessors.

Chapter 2

CODING AND MODELING CONCEPTS

This chapter explains the precise relationship of stochastic processes to data compression, and prepares the reader for critical understanding of on-line modeling algorithms. In data compressors that use an explicit statistical model, the model is only loosely coupled with the remainder of the data compression machinery—the coder. So, one goal in explaining this relationship is to communicate why the model designer can forget about the data compression machinery itself, once he understands how it uses models. Enough basic information theory is presented to illuminate the natural, minimally dependent subproblems of data compression, namely source coding and source modeling. Then, the goals of optimal source modeling and coding are outlined and justified based upon those basic information-theoretic principles. Since source coding has already been solved optimally with arithmetic coding techniques, I divide the presentation of arithmetic coding into what any source model designer must know about arithmetic coding, and an optional section containing the internal details. After this chapter, the discussion focuses solely on statistical modeling issues.

2.1 The Modern Data Compression Paradigm

The goal of data compression is to assign a minimal decodable code to a sequence in a practicable way, given the (estimated) symbol probabilities of the sequence symbols. The goal of *on-line* data compression is to do so in a single pass over the sequence, with no prior knowledge about the sequence or its symbol probabilities. Thus, the problem of on-line data compression can be broken into two completely independent subproblems: estimate the symbol probabilities, and code the symbols in as many bits as the negative logarithm of the symbol probabilities. The latter problem, on-line coding, has been solved optimally with arithmetic coding (see Section 2.3).

The remaining problem in on-line data compression is data modeling for probability estimation. A model may be considered in two logical parts: its *structure,* which organizes the set of events (alphabet symbols) modeled such that each event is

associated with a context (surrounding alphabet symbols) in which it has occurred; and its *parameters,* which are the conditional probabilities of the events given the contexts in which they occur. Universal models deduce, from the source message, a model which is a close approximation to the source that presumably emitted the message. Therefore, universal on-line models are doubly adaptive; both the structure and the parameters change to reflect the properties of the data.

The separation of modeling from coding, and the use of doubly adaptive stochastic models is the *modern data compression paradigm*,[1] pioneered by Glen Langdon and Jorma Rissanen [RL81, LR83, Ris83], and further developed in [Ris86a, MGZ89]. By "paradigm," I mean an ideal and general blueprint for other solutions to follow, and Rissanen's and Langdon's certainly fits this strict interpretation. As mentioned earlier, this paradigm is fully general: it is widely applicable, and it includes most if not all other types of solutions as special cases. It is ideal for a number of reasons: it divides the problems into subproblems which are minimally dependent, and it is rigorously based upon principles from information theory.

2.2 Entropy and Stochastic Processes

The goal of this section is to explain why *optimally* compressing a sequence $a_1 a_2 \cdots a_n$ requires first deducing (or knowing) the stochastic FSM S that presumably emitted the sequence and then encoding the individual symbols a_i of the sequence using codes of length $-\log P(a_i|S)$. This fundamental result of information theory is the basis for the separation of coding and modeling in statistical data compressors. Much of the terminology introduced in this section will be used repeatedly throughout the thesis.

The **entropy,** H of a single random variable X, with probability distribution p over an alphabet A is defined as

$$H(X) = -\sum_{a \in A} p(a) \log(p(a)). \tag{2.1}$$

A **stochastic process** is a sequence of random variables, with arbitrary dependence among them. If the dependence between successive random variables is limited to the preceding random variable, the sequence is *Markovian.*

---

[1] This is Ross Williams' terminology [Wil91].

A **Markovian FSM information source** is a finite-state machine (FSM) representation of a Markovian stochastic process such that each state corresponds to a random variable and for any state there is at most one transition for each output symbol, or *source symbol* in the FSM alphabet, and each transition is assigned a probability (e.g., see Figure 2.1.a). The probabilities of the transitions exiting any given state i sum to unity and therefore form a probability measure. That distribution is said to be *conditioned by the state i.*

In contrast, a **non-Markovian FSM information source** is a similarly defined FSM whose states may have more than one outgoing transition for a given output symbol, such that the probability of the ith transition depends on more than the ith state and the ith source symbol. An example of this can occur with models such that the probability of the ith source symbol is a function of only the current state, but in which the destination of the transition taken on the ith source symbol is a function of what states were visited prior to the current state (see Figure 2.1.b). Thus, the next state may depend upon an arbitrarily long sequence of previous states. Here the probabilities of the transitions conditioned by any particular sequence of preceding states will sum to unity.

When in a given state, an FSM source will randomly select an out-transition according to the state's output-edge distribution. When any transition is taken, that transition's associated alphabet symbol is output, and the source is said to have *predicted a source symbol.* The inputs to a source can be viewed as random numbers in (0..1], and the outputs are source message symbols from a finite alphabet A. In contrast, the inputs to an **FSM model** are symbols from A, and the outputs are numbers in (0..1], that is, symbol probability estimates. When an FSM model is in a given state s, it will produce a probability estimate of the currently scanned source message symbol using the current state s's output-edge distribution.

The **entropy of a source** S is given by

$$H(S) = \sum_j P_j H_j, \tag{2.2}$$

where H_j, the entropy of the distribution of the j^{th} state, is computed using equation 2.1, and P_j is the probability of being in state j at any time. For *ergodic* sources the P_j's correspond to an eigenvector P with eigenvalue 1 of the transpose T' of the asymptotic transition matrix T (i.e., $P = T'P$). For *stationary* sources, the P_j's do

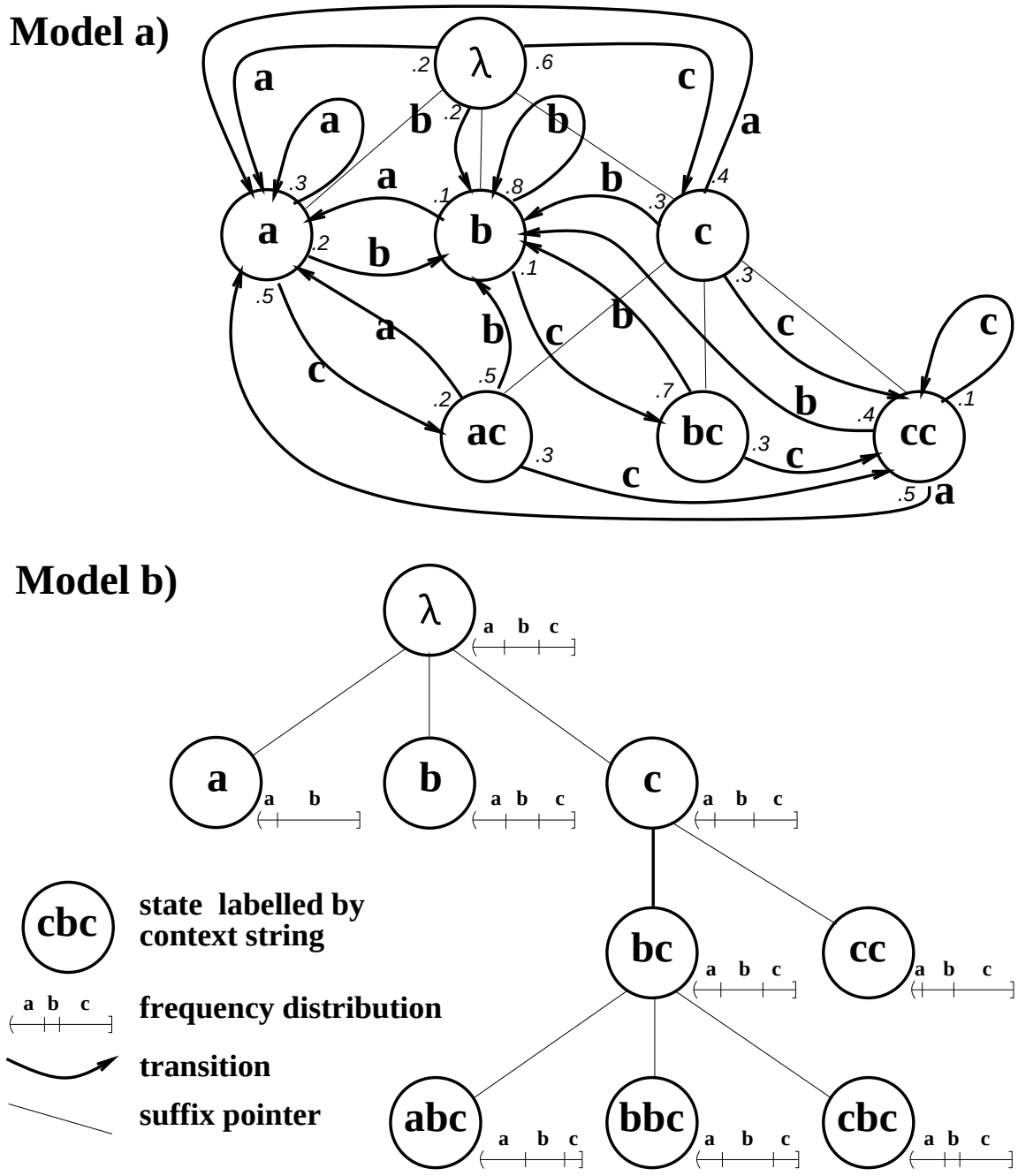

Figure 2.1: **Examples of Markovian and Non-Markovian FSM Sources.** Each state is labeled with a regular-set description of the strings that may bring the machine into that state. The regular-set descriptions are implicitly concatenated on the left with A^*. Model 2.1.a is Markovian. The transition function of Model 2.1.b is not shown—it has a closed form that forces the model into the state corresponding to the maximal matching suffix of the already-processed portion of the source message. Model 2.1.b is not Markovian, since state 'b' may go to state 'bc' or state 'abc' on symbol c, depending on which source symbols preceded state 'b'. Note that both models shown are FSMX (discussed in Section 2.5), since their next-state functions always go to the state corresponding to the maximal matching suffix of the already-processed portion of the source message.

not depend on the initial state.

An **ergodic source** is the most general dependent source[2] for which the strong law of large numbers holds [CT91, Chapter 15.7]. That is, if an ergodic source is allowed to emit symbols indefinitely, the relative frequency of the symbols, given the state the source is in when it emits the symbols, will converge to the transition probabilities of the source. A formal definition of *ergodic* is beyond the present scope, as is most formal probability theory. However, the ergodicity of an FSM source has been proved to be a decidable graph-theoretic property [Paz71, Sha48]. An FSM is ergodic if the graph consisting of its states and transitions that have non-zero probability meets both of the following conditions:

1. The graph is strongly connected, i.e., every state is reachable from every other.
2. The greatest common divisor of the set of integers consisting of lengths of all distinct circuits in the graph equals one. (Note that this set is closed under addition.)

A **stationary source** is one in which the probability distribution that selects the initial state is time-invariant. This implies that the output of the source does not exhibit different statistical properties due to *phase shifts*. That is, for all i and k, the infinitely long sequence $a_i a_{i+1} \cdots$ has the same statistical properties as the sequence $a_{i+k} a_{i+k+1} \cdots$, if they are both emitted by the same stationary source.

In his "noiseless source coding theorem" [Sha48], Shannon proves that assuming a finite, stationary, and ergodic source, members of the set of messages that can be emitted by that source cannot be uniquely encoded with fewer output symbols per source character than is given by the source's entropy.

Therefore, the **minimal codelength of a string**, $a_1 \cdots a_n$, with respect to an (ergodic, stationary, and finite) information source S is given by

$$H(a_1 \cdots a_n | S) = -\frac{1}{n} \sum_{t=1}^{n} \log(P(a_t | s_t)), \tag{2.3}$$

where a_t is the t^{th} symbol of the string, and s_t is the state of S at time t. Thus the minimal codelength of a string with respect to a source equals the expected per-symbol minimal code length that can be assigned by that source.

---

[2] Dependent sources include stochastic sources and models, as presented here.

We would like to be able construct *uniquely decodable* codes for a given source, whose lengths are close to the source's entropy. Any uniquely decodable code satisfies the **Kraft-McMillan inequality**:

$$\sum_{i=1}^{N} 2^{-l_i} \leq 1,$$

where the l_i are the code lengths in the given code's alphabet, and N is the size of the code alphabet. Conversely, given a set of code lengths that satisfy this inequality, it is possible to construct a uniquely decodable code with their lengths [CT91, chapter 5].

Shannon's coding theorem and the Kraft-McMillan inequality reduce the problem of optimal source coding to that of computing a uniquely decodable code with code lengths equal to the (finitely represented) negative logarithm of the source probabilities.

2.3 Optimal Source Coding

There are many efficient and provably optimal solutions to the source coding problem, that is, the problem of encoding symbols in a number of bits equal to (or arbitrarily close to) their minimal codelengths. All can be viewed as variations of *arithmetic coding.* With no loss of generality, arithmetic coding is performed one source symbol at a time; it only requires an estimated probability of the currently scanned symbol. The symbol-wise probabilities required for on-line coding are obtained in the modeling process, which in one pass constructs an adaptive finite-state model.

Arithmetic coding [Pas76, Ris76, RL79, Rub79, Gua80] codes each symbol with a number of bits that can be made arbitrarily close to the negative logarithm of the symbol's probability, by increasing the size of the registers used to do the arithmetic. Note that this implies that some symbols must be encoded with a fractional number of bits. Generally speaking, this is achieved simply by combining adjacent symbols until their combined probability estimate is a power of two, or very nearly so, and then emitting the code.

Arithmetic coding, and how it interfaces with models, is notoriously difficult to grasp. The following presentation and its illustration in particular are a departure from the usual descriptions, exemplified by [RL79] and [WNC87]. We first give an operational description that steps through how a stochastic model and an arithmetic

coder/decoder work together to encode and then decode each symbol of a source message. Then we explain the elegant internal operation of an arithmetic coder in an optional section that can be skipped without loss of continuity. Lastly, we reiterate the minimum that every potential model designer must remember in order to correctly design stochastic model suitable for on-line lossless data compression.

2.3.1 On-line Coding and Decoding: Step by Step

The interface between arithmetic coding and stochastic models is pictured in Figure 2.2, where coding is pictured as executing the following five labeled steps for each message symbol:

1. Estimate the current *coding distribution*, that is, the cumulative probability distribution in $[0 \ldots 1)$ over the input alphabet A, conditioned by the current state of model $M(a_1 a_2 \cdots a_{i-1})$. Note that the coding distribution maps each symbol in A to a unique, non-overlapping subinterval of $[0 \ldots 1)$.
2. Read in the currently scanned source symbol a_i (e.g., a_{12} = 'a') and compute its probability estimate using the coding distribution.
3. The probability estimate is represented as the endpoints of the scanned symbol's subinterval in the current coding distribution. The coder uses these endpoints, shown as dashed lines entering the arithmetic coder, to construct the *codepoint*, illustrated as a heavy line exiting the arithmetic coder and then entering and exiting the arithmetic decoder. The codepoint is the number in $[0 \ldots 1)$ that is represented by the bits that are transmitted to the decoder, high-order bits first.
4. The current source message symbol a_i is then sent as input to a mechanism that adapts model structure and parameters.
5. Lastly, the current source message symbol and model state are used to determine the next current state.

Decoding proceeds in 5 similar steps, also pictured in Figure 2.2:

1. Same as Coding Step 1.
2. The codepoint of the final message falls in a particular interval of the current state's cumulative distribution within $[0 \ldots 1)$. The already-received high-order bits of the codepoint provide an approximation of the final codepoint and fall

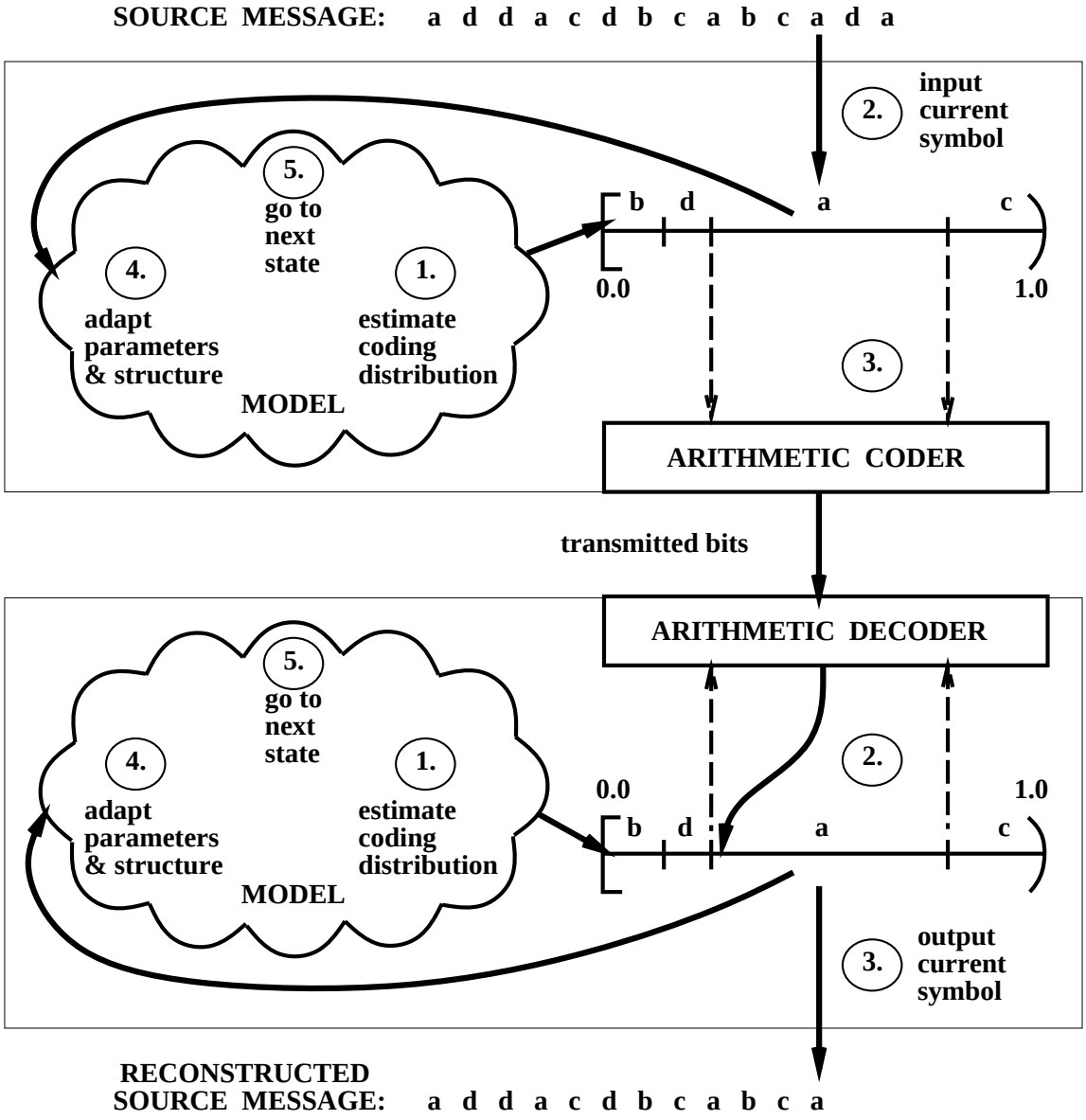

Figure 2.2: **The Coder/Model Interface.** Coding and decoding execute the five labeled steps for each message symbol. The concatenation of the transmitted bits for the entire source message, in reverse transmission order, represents a binary fraction in $[0 \ldots 1)$ called the *codepoint*. Note that only steps 2 and 3 differ between coding and decoding.

into the same interval of the current state's cumulative distribution as the final codepoint would. The subinterval of the current coding distribution that contains the codepoint is determined, and its endpoints, shown as dashed lines into arithmetic decoder, are sent to the arithmetic decoder so it can update (and renormalize) its coding interval.

3. The source message symbol corresponding to the subinterval of the current coding distribution that contains the codepoint (e.g., $a_{12} =$ 'a') is output.
4. Same as Coding Step 4.
5. Same as Coding Step 5.

2.3.2 Inside Arithmetic Coding (Optional)

The internal operation of an arithmetic coder revolves around computing and transmitting the *codepoint*. Arithmetic coding was derived from Elias coding, which computes an infinite-precision codepoint.

Elias Coding

The *codepoint* can be any number in the *coding interval* $[L \ldots L + W)$ in $[0 \ldots 1)$, where

- $W = \prod_{j=1}^{i} p_j$ is the width of the coding interval after a_i has been coded.
- $p_j = P_e(a_j | M(a_1 a_2 \cdots a_{j-1}))$ is the width of a_j's subinterval in the coding distribution estimated by $M(a_1 a_2 \cdots a_{j-1})$.
- $L = \sum_{j=1}^{i} (Q_j \prod_{k=1}^{j-1} p_k)$ is the low endpoint of the coding interval after a_i has been coded, and
- Q_j is the low endpoint of the a_j's subinterval in the coding distribution estimated by $M(a_1 a_2 \cdots a_{j-1})$.
- Computing L and W iteratively for each source symbol a_j is simple:
  Initially, $L = 0$ and $W = .999999\ldots$
  for $j \geq 1$ **do begin**
  $L \leftarrow L + Q_j \cdot W$;
  $W \leftarrow W \cdot p_j$;
  end

Elias Coding is Optimal

Without loss of generality we can assume that the codepoint equals L. Thus, L is transmitted to the decoder. Conceptually, the codepoint is not completely defined until the entire message has been coded. The number of bits required to represent the final codepoint L equals the number of bits from the decimal point to the least-significant non-zero bit of W. After a_n has been processed, W equals

$$\prod_{i=1}^{n} P_e(a_i|M(a_1 a_2 \cdots a_{i-1})).$$

So, the number of bits transmitted to send the codepoint L equals

$$\sum_{i=1}^{n} -\log P_e(a_i|M(a_1 a_2 \cdots a_{i-1})).$$

Incremental Transmission

As described above, computing the endpoints of the coding interval requires infinite precision. However, data compression practice requires the qualities of incremental message transmission and finite representation of state variables. These implementation goals are are achieved simultaneously by transmitting the codepoint L incrementally.

Consider the binary representations of L and W, where X denotes a "don't care" bit value:

$$\begin{aligned} W = \; & .\overbrace{0\ 0\ 0\cdots 0}\ \overbrace{000\cdots 0}\ \overbrace{1\,XX\cdots 1}^{significant}\ \overbrace{00\cdots 0}^{zeros} \\ L = \; & .\underbrace{XXX\cdots X}_{transmittable}\ \underbrace{011\cdots 1}_{unstable}\ \underbrace{XXX\cdots X}\ \underbrace{00\cdots 0} \end{aligned}$$

As the coding interval narrows with each source symbol, a smaller proportion of the bits of W remain significant. If we multiply L and W by the same constant during coding, the output code is unchanged. Thus, we can shift out some of the high-order bits of L as long as they are stable, they correspond to insignificant bits of W, and we shift W by as many bits. Stable bits are those that will not change; unstable bits can be affected by a carry from the lower order bits of L. The highest order unstable bit is 0, the rest are 1s. The process of shifting out the high-order bits of W and L is called *renormalization*. It scales up the coding interval and incrementally transmits

the high-order bits of the codepoint L. Internally, renormalization is equivalent to moving the decimal point of W and L to the right:

$$\begin{array}{rcccc} & & & & \overbrace{}^{significant} \\ W \propto & \overbrace{0\ 0\ 0\cdots0}^{} & . & \overbrace{000\cdots0}^{} & \overbrace{1\,XX\cdots1}^{} \\ L \propto & \underbrace{XXX\cdots X}_{transmitted} & . & \underbrace{011\cdots1}_{unstable} & \underbrace{XXX\cdots X}_{} \end{array}$$

Fixed Precision

Note that incremental transmission as described above does not achieve fixed precision, for there may be an arbitrary number of unstable bits. Fixed precision may be accomplished by fixing the number of significant bits that represent W (and thus limiting the minimum allowed width of a symbol subinterval in a probability distribution) and renormalizing W and L so that the decimal point always falls just before the significant bits of W. The stable bits are transmitted, but the unstable bits are merely counted, until they too become stable:

$$\begin{array}{rcccc} & & & & \overbrace{}^{W_{\text{renormalized}}} \\ W \propto & \overbrace{0\ 0\ 0\cdots0}^{} & \overbrace{000\cdots0}^{} & . & 1\,XX\cdots X \\ L \propto & \underbrace{XXX\cdots X}_{transmitted} & \underbrace{011\cdots1}_{counted} & . & \underbrace{XXX\cdots X}_{L_{\text{renormalized}}} \end{array}$$

Note that $\frac{1}{2} \leq W_{\text{renormalized}} < 1$.

Unstable bits cannot be transmitted, but they can be counted instead of kept in a register. We keep track of the number of unstable bits of L as follows:

- If the computation of the next coding interval carries into the unstable bits, they will all be complemented and all but the last bit, now a 0, will become stable. Transmit the stable bits of L represented by the unstable bits counter and reset the counter to 1.
- If the renormalization of the coding interval shifts out a 0 bit, transmit the unstable bits of L represented by the unstable bits counter, and reset the counter to 1.
- If the renormalization of the coding interval shifts out a 1 bit, increment the unstable bits counter. If the unstable bits counter reaches its maximum value, force stability upon the unstable bits of L that it represents by transmitting

them all, and then resetting the unstable bits counter to 1. Here, resetting the unstable bits counter inserts an extra 0 into (the untransmitted and unstable portion of) the bit stream, and is called "bit stuffing" [RL79]. An alternative to bit stuffing can be found in [WNC87].

In summary, a fixed-precision arithmetic coder incrementally transmits an *approximate* codepoint that is larger than the exact codepoint. There are two determinants of coder accuracy. It is most affected by the number of bits p used to represent the width of the renormalized coding interval, $W_{\text{renormalized}}$. Accuracy is also affected by the maximum value of the unstable bits counter, but to a lesser extent.

2.3.3 The Least *a Model Designer Should Know*

Model designers need not concern themselves with the mechanics of arithmetic coding. Instead, they need only know the coder's properties and the required interface between the coder and the data model.

For each given input symbol, the model estimates a probability distribution over an m-ary alphabet, where the alphabet symbols are in a given but arbitrary order. The model sends relevant points in that probability distribution to the arithmetic coder, and afterwards, the input symbol is used by the model to perform updates. During an update, both model structure and model parameters may change. The list below completely describes what the model designer must know about arithmetic coding. We assume that the current coding distribution for each source symbol is given as a cumulative frequency distribution.

changeable coding distributions: The coding process incurs no extra cost if the ordering of the alphabet symbols or the current coding distribution completely change for each symbol coded. As long as the changes are deterministic and based on the past and current input symbols, the decoder's model can track those changes after it decodes the current symbol.

cumulative frequencies: Assuming the entire mass of the coding frequency distribution is known, the encoder need only know the cumulative frequencies of the symbols preceding the current symbol according to the arbitrary symbol ordering, and the frequency of the current symbol.

approximate symbol intervals: When a symbol's interval in the coding distribu-

tion is approximated, ensure decodability by using a higher or equal value of the low end of its interval, and a lower or equal value of the width of its interval.

precision: Coder implementations are of limited precision, which is determined by the number of significant bits p in the representation of the coding interval. The minimum allowed estimated probability of a symbol is 2^{1-p}. The simplest solutions scale by half all counts in a frequency distribution if the total frequency count on the transitions leaving a node reaches a maximum frequency limit of 2^{p-1}. Scaling also helps models accommodate non-stationary behavior of data.

special 'stop' symbol: Arithmetic decoders must be told when to stop, lest they use the codepoint to 1) output the symbol according to the codepoint's location in the current coding distribution, and 2) take a transition to a new state, and so on, indefinitely.

binary is simplest: Arithmetic coding can be greatly simplified if the alphabet size, m, is always two. Sometimes that can be conveniently arranged for a model that uses a larger alphabet, via alphabet decomposition.

The key benefits of arithmetic coding are that it imposes no restrictions on the model and it encodes symbols using (fractional) code lengths that can be made arbitrarily close to the symbol's theoretically minimal codelength. Implementations of arithmetic coding have been honed to near perfection over the years. There are now several fast and extremely accurate approximations which eliminate the divisions and multiplications that are required by straightforward implementations [RM89, CKW91, FGC93].

2.4 A General Strategy for Optimal Source Modeling

Assume the string was generated by a stochastic FSM.

Given a string to compress, we first assume that it was generated by a finite stochastic source belonging to a tractable but useful class of models. The goal in universal coding is to deduce a finite stochastic source for the input string that maximizes the string's likelihood, and thereby to minimize the expected code length with respect to that source. This process is more tractable when it is assumed that the original message source is *stationary* and *ergodic*. These two properties allow us to assume that any

long-enough substring of the source string fully represents the source's probabilities.

Use local context to condition each symbol's probability estimate.

The models constructed when performing on-line source modeling are finite-state machines whose states are associated with a frequency distribution over the output edges, which are labeled by the possible next symbols. In many models, each state is also associated with a string of symbols from the input alphabet, called a *conditioning context.* For each sequence symbol, the state with the longest context that is a suffix of the already scanned portion of the input sequence becomes the current state of the model. Each state's frequency distribution is conditioned by its context. It is from these distributions and the current FSM state that the (conditioned) probability estimate of each sequence symbol is computed and then sent to the coding unit.

Construct the model incrementally from a single state.

On-line models are constructed adaptively, usually from a trivial initial model, with no *a priori* knowledge of the input string. Adaptations increment context-conditioned frequency counts (i.e., *state parameters*) and periodically alter the structure of the model by adding new states, after each symbol is processed. Decodability is ensured by forcing model structure, and the probability estimations it produces, to rely only upon previously coded input. Thus, the decoder's model tracks the encoder's model as it recreates the original input string. More precisely, for any given source message, after the i^{th} input symbol is processed by the coder, and similarly after the i^{th} output symbol is decoded by the decoder, the model is adapted deterministically using the i^{th} symbol. Thus the coder model and decoder model are step-wise identical. The logical synchronization of the coder and decoder models is pictured in Figure 2.2.

Adapt the model aggressively...

To deduce a source model on-line, we build a model so that the entropy of the model with respect to the string-seen-so-far decreases as more and more of the string has been processed. This is best described as "fitting the model to the string." By doing so, we strive to minimize the code length of the unprocessed portion of the string. And if the model fits a long-enough source substring string well enough, we can assume

we have deduced the source. If the source is ergodic, and the long-enough string is a proper prefix of the string to be compressed, it can achieve the optimal compression for the remainder.

...but be prepared to detect and roll back over-adaptation.

This may seem insidiously cyclic. One problem is that we never know what "long enough" is, because that is a function of the unknown source. If a growing model is tailored to fit a string that is not long enough to represent its source, the model is said to be *over parameterized* and may perform poorly on the rest of the string. Waiting too long to fit the model discards opportunities to code with shorter code lengths. In most implementations, that means that useful source statistics are discarded as well. Later in this thesis, it will become clear that a good solution is to adapt the model aggressively, but in such a way that over-parameterization is dynamically rolled back as needed. That is, the model should be tailored early and often, so that at any time, if it were rerun on the portion of the source string from which it has been built, it would assign a high probability to that portion of the string.

2.5 A Hierarchy of Important Model Classes

When trying to deduce an assumed source model from a source message, we usually exclude stochastic sources of the fullest generality. This section presents the subclasses of stochastic sources that appear often in the literature. A clear understanding of the class containment relationships is important for comparing the relative power of different modeling techniques, especially in the face of statements that claim an algorithm's *asymptotic optimality,* which is necessarily *with respect to an assumed class of information sources.* The more restricted the assumed class, the weaker the result.

Three interesting subfamilies of stochastic FSM sources from the literature are *Markov sources* (Section 2.2), *FSMX sources* [Ris86a], and *finite-order FSM sources* [Ash65], which are also called *finite-context automata* [BM89]. The containment relationships between these model classes are shown within the Chomsky Hierarchy [HU79] in Figure 2.3.

- In **Markov sources**, the next state is completely determined by the previous

state and the current source symbol. That is, a model with state set S and input alphabet A is Markovian if, for any given state $s \in S$ and input event $y \in A^+$ (note that Markov models can have string transitions), the next state is only dependent upon s and y.

- In **finite-order FSM sources**, the next state is completely determined by testing a finite portion of the end of the already-processed portion of the source message for membership in a finite set of strings (i.e., conditioning contexts), that are associated with each node. More formally, a model with state set S is *finite order* if, for any state $s \in S$ and any input sequence $a_1 a_2 \cdots a_i$, s is the unique current state at time i if and only if $a_1 a_2 \cdots a_i \in A^* F(s) \cup G(s)$, where $F(s)$ and $G(s)$ are finite sets in A^*.
- **FSMX sources** are finite-order FSM sources such that only *singleton* sets of conditioning contexts are associated with each state, which we denote with the mapping $\mathbf{context} : S \rightarrow A^*$. Furthermore, for each state s there exists a state p such that $\mathbf{context}(s) = b \cdot \mathbf{context}(p)$, for some $b \in A$. That is, in FSMX models, the context strings of the states have single-symbol minimal extensions.
- There exist Markov sources that are not FSMX but which are, nonetheless, interesting in practice, namely DMC and several DMC variants that are introduced in Chapter 6. In Chapter 4, I prove that DMC states cannot be uniquely characterized by single conditioning contexts, and are therefore not FSMX.

Another common assumption made for Markovian and non-Markovian sources is that they are *unifilar*. A **unifilar Markovian FSM** has exactly one transition for each source symbol of non-zero probability leaving each state [Ash65]. More generally, an FSM source with "next-state" function $f : S \times A \times A^* \rightarrow S^*$ is *unifilar* if for all input sequences $a_1 a_2 \cdots a_n \in A^+$, all $i, 1 \leq i \leq n$, and all states $s \in S$, $|f(s, a_i, a_1 a_2 \cdots a_{i-1})| = 1$. Thus, the next state and source symbol are always completely determined by the already scanned portion of the input sequence and the current state's frequency distribution over the input alphabet. FSMX models are examples of unifilar sources.

The model hierarchy described in this section contradicts several statements from influential sources in the literature: many prominent researchers seem to labor under the impression that the class of FSMX sources properly contains the class of Markov sources, and that the class of unifilar, ergodic, finite-order sources equals the class of

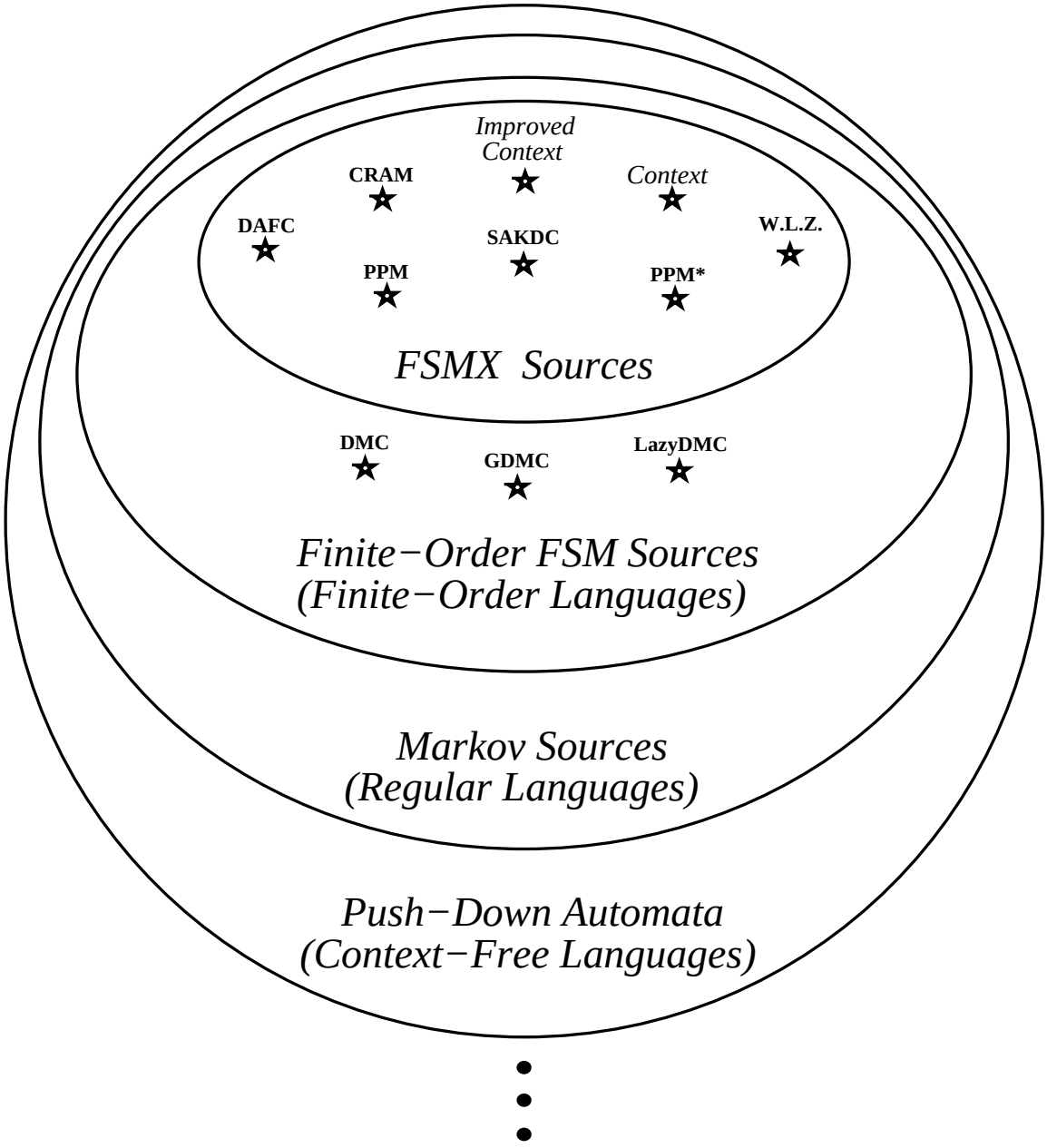

Figure 2.3: The relationships between the classes of languages generated by useful classes of information sources, plus the models built by some of the popular algorithms that influenced this work. All models shown are additionally unifilar and ergodic. The proper containment of languages generated by FSMX Sources in the class of languages generated by Markov Sources is proved in Chapter 4. The modeling algorithms plotted here are explained in Section 2.7.

FSMX sources. For example, the authority responsible for defining the FSMX class introduces the distinction in [Ris86a] as *"a useful subclass of FSM sources which includes the Markov sources,"* and later claims that *"This class, which we denote FSMX, generalizes the class of Markov sources."* Recently, the following description appeared in [WST95]: *"These [FSMX] sources form a subclass of the set of sources defined by finite state machines ('FSMs') and an extension ('X') of the class of finite-order Markov sources."* The authors of [WLZ92], give a definition of a class of sources that corresponds to the above definition of finite-order FSM sources and claims that, with the additional restrictions of unifilarity and ergodicity, the resulting class equals the class of FSMX models. All of these statements are incorrect: The class of models that can be generated by transition redirection, exemplified by the DMC algorithm and analyzed in Chapter 4, are unifilar, ergodic, and finite order, but they are not FSMX. Furthermore, DMC models can emulate FSMX models (although extra states will be required) and can thus generate the languages of FSMX models. However, as we prove in Chapter 4, there are languages generated by DMC models that cannot be generated by any FSMX source. Lastly, note that that the existence of finite-order models that are not Markovian does not contradict the containment of languages generated by FSMX models in the class of languages generated by finite-order Markov models. With additional states, a Markov model can emulate an FSMX model. Hopefully this discussion, summarized in Figure 2.3, clears up any reigning confusion about the class relationships.

On a final note, there is considerable interest in moving up the Chomsky Hierarchy by applying more powerful computational models to sequence prediction problems such as data compression. The implicit assumption that a sequence of English text was generated by a Markov or FSMX source, or even a push-down automaton, is poorly justified. Yet current on-line modeling algorithms construct source models with less generative power than the class of regular languages. Chapter 4 expands the known frontier of occupied territory within the class of regular languages by proving that there exist effective universal on-line models (i.e., DMC and LazyDMC) that are outside the FSMX class. However, as illustrated in Figure 2.3, the question of how to use the full generative power of (ergodic) finite state machines in an adaptive, universal, on-line model is still open.

2.6 Source Modeling with Suffix Trees

The concept most essential to effective model design and implementation is the relationship between the model structure and the conditioning context partition on A^* that is induced by the model's transition function and frequency update mechanism. Many of the model classes described in Section 2.5, and all models explored in the rest of this thesis, can be implemented as *suffix trees*. Suffix trees allow

1. distinct, competing models to be represented economically by embedding them in a single tree; and
2. a straightforward mapping between each model's states and the elements of its conditioning context partition.

This section introduces on-line modeling with suffix trees and the terminology that will be used throughout this thesis in four essays that cover: basic notation and terminology, the restrictions that model class place upon transition function implementation, the first in a series of primers on suffix-tree model semantics, and lastly, how to perform probability estimation on-line with a suffix-tree FSM.

2.6.1 Notation and Terminology

Broadly speaking, the suffix-trees used in on-line string modeling are finite-state machines, where the current state has an associated conditioning context string that is a suffix of the already-processed portion of the input sequence, and the edges leading out of the current state state correspond to the possible values of the next scanned source symbol. Moreover, each state's set of out-edges corresponds to a frequency distribution over the set of single input symbols that label them. The current state's frequency distribution is used to assign a probability to the next scanned input symbol.

For our purposes, a suffix-tree FSM is set of states such that each state s has access to the following information:

- A conditioning context string $\mathbf{context}(s) \in A^*$, where A denotes the input alphabet.
- A parent state $\mathbf{suffix}(s)$, such that $\mathbf{context}(s) = y{\cdot}\mathbf{context}(\mathbf{suffix}(s)), y \in A^*$, where the string y is known as the *minimal extension* to $\mathbf{context}(\mathbf{suffix}(s))$, and s is said to be a *child* of $\mathbf{suffix}(s)$.

- A list of out-events $a|s$, for $a \in A$, where each out-event $a|s$ is associated with a record, $\mathbf{count}[a, s]$, of the frequency of $a|s$ in the previously processed input.
- Its Markov *order*, which equals $|\mathbf{context}(s)|$.

All on-line models are grown incrementally from an initial suffix-tree model. The initial suffix-tree has two states s_0 and s_{-1}, where s_0 has no out-transitions and $\mathbf{context}(s_0) = \lambda$. State s_{-1} equals $\mathbf{suffix}(s_0)$, has no suffix (i.e., $\mathbf{suffix}(s_{-1}) = null$), and has out-events for every $a \in A$ that enter s_0, plus an additional event that is used to signal the end of the input sequence.

For general suffix-tree FSMs, call the language of the subtree rooted at state s $L(s)$. $L(s)$ equals the set of strings that take model M to state s or to any state in the subtree rooted by s. In general suffix-tree FSMs, for a given input history sequence $a_1 a_2 \cdots a_{i-1}$, the set of states $\{s : a_1 a_2 \cdots a_{i-1} \in L(s)\}$ is referred to as the *excited states*. If we view the suffix-tree as a single FSM, the *current state* is always the maximum-order excited state. For example, refer to Figure 2.1: after input sequence '*...abcbbc*' the current state of Model b) will be the state labelled '*bbc*' and the set of excited states will be labelled with context strings '*bbc*,' '*bc*,' '*c*,' and 'λ.'

Since the models we construct are stochastic, we must associate various quantities with model states and out-events. We shall employ the following convention. When the range of a mapping, say, f, can be described using a closed form, we shall use the function notation $f()$. However, when the range of the mapping involves a set of persistent variables that change state over time, such as the frequency counters that are associated with each transition, we shall use the notation $f[]$.

2.6.2 Suffix-Tree Model Families

For models such as PPM and PPM*, the FSM transition function is defined in terms of the function **context**() (e.g., see Figure 2.1.b). More generally, the transition function may not have a simple closed form, and may only be describable as explicit state-to-state pointers that are labeled by single symbols (e.g., see Figure 2.1.a). The abstract family membership of a modeling technique directly affects how its transition function may be represented in a computer, and therefore how the computation of the next state may be implemented.

The suffix trees underlying all PPM variants, including PPM*, are *Markovian FSMX models*. They are Markovian because they satisfy the *Markov Property*: for

any given state s and input event[3] $y \in A^+$, the next state is only dependent upon s and y. They are FSMX because the following hold for all states s:

- $L(s) = A^*\mathbf{context}(s)$, and
- the context string of s extends the context string of $\mathbf{suffix}(s)$ by exactly one symbol, that is,

$$\mathbf{context}(s) = b \cdot \mathbf{context}(\mathbf{suffix}(s)), \text{for some } b \in A.$$

The substring b above is called a *minimal extension* of $\mathbf{context}(s)$. Both models shown in Figure 2.1 are FSMX, since their next-state functions always go to the state corresponding to the maximal matching suffix of the already-processed portion of the source message.

The fact that the improvements developed and tested later in this work apply to FSMX models is important because FSMX models are ubiquitous in the information theoretic literature. FSMX models [Ris86a], are suffix-tree context models with single-symbol minimal extensions such that the next state given by the transition function on a given state and input symbol can have Markov order that differs arbitrarily from the order of the given state. By contrast, in a Markov model, the order of the next state given the current state s and input event $y \in A^+$ is always bounded by $|\mathbf{context}(s)|+|y|$. FSMX models are not always Markovian because for a given state s and input event y, the next state's order may exceed $|\mathbf{context}(s)|+|y|$. Thus, information about the prior input sequence which is not recorded in state s is required to determine that next state. For example, the model in Figure 2.1.b not Markovian, since state *'b'* may got to state *'bc'* or state *'abc'* on symbol c, depending on which source symbols preceded state 'b'.

The transition function of Markovian FSMX models may be implemented in one of two ways: The Markov property allows representation via explicit next-state pointers, while FSMX transition functions may be represented using a closed-form description. The closed-form description requires that the next state be computed using conditioning context extensions and children pointers, which must be stored at each state, plus an input history buffer. (The children pointers are best represented as a

[3] Models are commonly thought of as having only symbol transitions, but some may include string transitions.

pointer to a list of children linked by sibling pointers.) The state whose conditioning context is the longest suffix of the already processed input sequence $a_1 a_2 \cdots a_{i-1}$ is located by searching root-to-leaf for a child of each successively visited state of order k with extension $b = a_{i-k}$. An explicit pointer representation of PPM*'s transition function is possible only because PPM*'s model satisfies the Markov property. This flexibility is not a feature of all Markov models with underlying suffix-tree structure. We have proved that there exist useful finite-context Markov models that are not FSMX [Bun96, Chapter 4]. Those models allow arbitrarily long extensions to state contexts and include Dynamic Markov Compression (DMC) [CH87] models as a special case. They require explicit destination pointers because their conditioning contexts cannot be described by a single string.

2.6.3 Model Semantics I: Conditioning Context Partitions

The suffix-tree structure of the models discussed in this work may have a number of mathematical interpretations. So far, we have described suffix-tree models as representing a single FSM model, with a single current state. However, in the most general interpretation, the suffix tree represents a set of distinct, nested, suffix-tree FSM models, that share the same transition function mapping $A \times S$ to S, where S equals the set of states (or nodes) in the suffix tree.

Each distinct FSM corresponds to a complete frontier of the tree rooted by the order zero state s_0, and *vice versa*. A *frontier* of s_j's subtree T consists of the leaves of a subtree of T, rooted at s_j. A tree T has as many distinct frontiers as it does distinct subtrees that share its root. A frontier of a subtree T rooted at s_j is *complete* if it consists of s_j, or if it consists of complete frontiers of the subtrees rooted by each of the children of s_j.

The states of each complete frontier of a suffix-tree FSM rooted at s_0 impose a distinct partition on the set of possible conditioning contexts in A^*. Each state s on a given complete frontier contributes an element of that frontier's conditioning context partition. The partition element of state s equals $L(s)$, the set of strings that cause any of the nested FSMs emulated by the suffix tree to make a transition into any state that is in the *subtree* rooted by state s. Although state s may simultaneously belong to several complete frontiers of a given suffix tree, s contributes the same partition element to each frontier's associated context partition. The partition elements of all

possible children of s form a refinement of s's partition element. For the FSMX class of models, every state's partition element equals the set of strings $A^*\mathbf{context}(s)$.

The single FSM interpretation corresponds to the maximum complete frontier of the suffix tree. We shall adopt the multiple-model interpretation from here on, but will continue to use the term "current state" as shorthand for maximum order excited state.

Note that regardless of the intended mapping between model states and conditioning context partition that is denoted by the context strings of the states, the true mapping is defined by which state's frequencies are updated given particular input histories (note that updating the frequencies of all excited states is only one of several ways to perform frequency updates). When the true mapping and intended mapping disagree, the (intended) context partition has been violated, and the predictive ability of the model is compromised.

2.6.4 On-line Probability Estimation with Suffix Trees

Consider the excited states in a suffix-tree model, for some arbitrary input sequence $a_1 a_2 \cdots a_{i-1} \in A^*$. We wish to estimate a probability distribution over all possible values of a_i. The estimated probability distribution over all symbols in A will be conditioned by the excited states, and therefore by the already-processed portion of the input sequence, which is the sole source of the frequency data organized at all model states, except s_{-1}. We will use the estimated distribution to assign a probability to the actual input symbol a_i. Thus, from the symbol frequencies conditioned by these states, we must estimate a single distribution that assigns non-zero probability to all symbols in the input alphabet A.

After each symbol instance a_i has been processed, new states may be added to the model as descendants of the maximum-order excited state. Lastly, the excited states, which now include any new states s such that $L(s)$ contains $a_1 a_2 \cdots a_{i-1}$, will each receive frequency updates for the symbol $a \in A$ such that $a = a_i$.

2.7 A History of Influential Suffix-Tree Models

Figure 2.3 places in the Chomsky hierarchy several algorithms from the literature, which we explain here in a mini-history of statistical data compression. First, a

glance at the bibliographies of most data compression papers reveals that there are two separate research communities. The first community, represented in information and coding theory literature, has produced the following constructions:

- The granddaddy of the suffix-tree stochastic modeling algorithms is the Doubly Adaptive File Compression algorithm, or DAFC [LR83]. DAFC introduced the now sacred notions of adapting model structure and frequency data simultaneously, and of separating coding from modeling in on-line data compression algorithms. Developed in the days of expensive memory, DAFC starts with a complete order-1 m-ary suffix tree, then dynamically adds and prunes a fixed number of order-2 states as needed. DAFC handles run-length subsequences separately.

- Rissanen's algorithm, *Context* [Ris83], aggressively grows an arbitrarily large binary-alphabet FSMX model. It adds all possible (there are only 2) refining children to a state as soon as the state's conditioning context occurs in the input sequence a second time, and then uses a top-down neighborhood-entropy-based heuristic for dynamically pruning the model. *Context*'s dynamic pruning technique, or "state-selection", was companion to some fundamental asymptotic results in sequential coding theory.

- The CRAM algorithm (Cohn and Ramabadran's Acronym Mystery?) [RC89] grows a binary FSMX model more cautiously. It employs a suboptimal state-entropy-based heuristic that adds minimal extensions to any state whose entropy is greater than some threshold value. The information-theoretic assumptions on which this model growth heuristic is based are flawed; even when the parameters of a given maximum order state in the source FSM have been completely deduced, the CRAM model will continue to add higher-order refinements to its version of the source state if the state's entropy lies above a given threshold. Moreover, slow and careful model growth is now known to have uncompetitive performance in practice—it is more effective to grow the model aggressively and then prune it back. However, the idea of generalizing DAFC models so that the model structure used for computing probability estimates was based upon model entropies was on the right track.

- There are later variants of *Context* that handle arbitrarily-sized m-ary alphabets using lazy evaluation of state refinements (e.g., [Ris86b]) and/or flawed[4] but efficient MDL-based[5] implementations of Rissanen's top-down pruning [Fur91]. In lazy evaluation of state refinements (i.e., children), only the refinement that is presently represented by the input sequence is added to the model, rather than adding all m possible refinements.
- A recent asymptotic result by Weinberger, Lempel, and Ziv (WLZ) [WLZ92] provides the first published construction with provably optimal performance, assuming an infinite input sequence and assuming that the FSM that presumably emitted the input sequence conformed to a known order bound. The authors make the key observation that *Context* systematically underestimates the local order. WLZ constructs a suffix tree with an order bound o and the property that for every substring w of $a_1 a_2 \cdots a_i$ such that $|w| \leq o$, there exists a state $s \in S$ such that $\mathbf{context}(s) = w$. WLZ is basically an order-bounded m-ary *Context* algorithm with maximally aggressive, lazily evaluated tree growth, combined with asymptotically optimal dynamic pruning.

In contrast to the coding theory literature, empirical data compression literature is dominated by a set of contributions that focus on *ad hoc* computation of probability estimates. Roughly speaking, these estimators "blend" statistics from different order models in a weighted average. "Blending" techniques have been tuned empirically over the years, primarily using the Calgary Corpus [BCW90]. Apparently, no empirical studies of the *Context* algorithm, its variants, or the WLZ algorithm, have been published, and it is not known how these algorithms perform on actual data. We interpret this as indication that their performance is generally not competitive; our own experiments (see Chapter 9) support this interpretation.

The Calgary Corpus has played a fundamental role in the development and acceptance of the algorithms below by allowing meaningful performance comparisons. However, its acceptance as the standard benchmark has led to the questionable practice of tuning allegedly "universal" algorithms *a priori* to the test data. The upside

---

[4] See Section 7.3 for an explanation of the flaw.

[5] Rissanen's work in coding theory led to his development of Stochastic Complexity [Ris89], which presents the Minimum Description Length or MDL principle, and which has since revolutionized the field of statistical inference. See Chapter 7.

is this: apparently the Calgary Corpus is a "good" benchmark, and only algorithms that implement sound heuristics and constructs can match, much less beat, the current competition. There is considerable variety among the 14 files of the Corpus and an algorithm that is tuned in advance to a subset of the files will do relatively poorly on the others, while an algorithm based upon sound reasoning that has competitive performance on some of the files will perform as well on the other files, compared to other algorithms. Lastly, performance on the Corpus tends to parallel performance on other data.

The algorithms below all grow very large suffix-trees and are not used in practice yet, because of their high memory and computation requirements. Today's data compression practice is dominated instead by fast string-matching techniques based upon Ziv-Lempel algorithms [ZL77, ZL78]. However, the trend in computer system tradeoffs continues toward increasingly cheaper computation and memory relative to secondary storage and communication bandwidth. The algorithms below represent the state of the art with respect to compression performance. Although performance depends greatly on the source message, on a typical mix of computer files, the best statistical techniques deliver better than 3:1 compression on average. This compares favorably with the the expected 2:1 compression of Unix `compress`, or the expected 2.5:1 compression of `gzip`, which are based upon [ZL78] and [ZL77], respectively.

- PPM [CW84b], or Prediction by Partial Matching, grows an m-ary alphabet FSMX model aggressively and on-demand, using lazy evaluation of state refinements and a fixed global order bound. PPM's reign as the best-compressing modeling algorithm has stood unchallenged for the 12 years preceding this thesis. PPM's success is due to its *ad hoc* "blending" technique for computing probability estimates, and the careful engineering behind Moffat's 1990 implementation, PPMC [Mof90].

- DMC [CH87], or Dynamic Markov Modeling, grows an unstructured binary-alphabet finite-state machine by redirecting any sufficiently popular transition to a new state in a process called "cloning." Miraculously, the maiden implementation of this simple idea delivered performance competitive with carefully engineered and tuned implementations of PPM. For years, practical data compression researchers have asked: Can DMC, given proper care and feeding, outperform PPMC? What does the context partition of DMC look like, anyway?

Are there hidden, abstract feature of DMC that could be useful in improving PPMC?

- SAKDC, or Swiss Army Knife Data Compression [Wil91], is a reimplementation of PPM with a suffix-tree and a vast number of justifiable but *ad hoc* features, none of which significantly improved PPM's performance. The well-organized empirical studies proved, above all else, that it will take more than bright ideas and thoroughness to outperform PPMC.
- GDMC [TR93], or Generalized DMC, is the first published[6] generalization of DMC to an m-ary alphabet that does not sacrifice compression performance or create an astronomically large model. GDMC does not, however, outperform PPM.
- PPM* [CTW95] does away with PPM's global order-bound in a linear-space implementation. However, despite its unbounded order, PPM* does not outperform higher-order implementations of PPMC.

Because of the relative lack of communication between the information-theoretic and empirical data compression researchers, and in spite of the steadfast hold that Moffat's 1990 implementation of PPM has had on the state of the art, the branches of the above history are laden with ripe, low-hanging fruit. The following chapters harvest that fruit, by formally answering the riddle of DMC, slashing PPM's memory requirements, improving blending, taking the order bounds and suboptimality out of state selection, and meaningfully combining blending and information-theoretic state selection for the first time.

2.8 Suffix-Tree Models in this Thesis

The rest of this thesis explores in detail the three most effective practical stochastic modeling techniques from the data compression literature: PPM, PPM*, and DMC. We develop techniques that generalize and improve the algorithms as a group. These algorithms were not originally introduced as having a common structure, and none were originally implemented or described with suffix trees. Yet, since several promising asymptotic techniques from the information-theoretic literature are based upon

[6] In [Bun94], I independently proposed a very similar solution, which I called "lazy cloning" and "LazyDMC," that leads to slightly better performance. It is the topic of Sections 6.5 and 9.5.

suffix trees, the next two chapters transform each of the algorithms to a suffix-tree representation and identify their semantics (i.e., the mapping between suffix-tree states and the conditioning-context partition). The correctness of the semantics of PPM and PPM* models is straightforward because they are trivially FSMX. However, the correctness of the semantics of DMC models requires formal proof, because, as we show, they are not FSMX. Once we have transformed these popular techniques to a unified (suffix-tree) structure, we systematically improve their performance.

Chapter 3

A MINIMAL SUFFIX-TREE IMPLEMENTATION OF PPM AND PPM*

Prediction By Partial Matching (PPM) has set the standard in lossless data compression research since its introduction in 1984 [CW84b]. PPM constructs a bounded-order Markov model such that the maximum order of the excited states is allowed to vary within the order bound. Recently, a new variant, PPM*, was developed that eliminated the (arbitrary) order bound in PPM's model structure [CTW95]. Both PPM and PPM* were developed without explicit description of their underlying suffix-tree structure, although several authors have since transformed PPM to suffix-tree-based implementations.

In this chapter, we transform PPM* so it will fit into an executable taxonomical framework that we introduce in Chapter 8 to decompose several other on-line modeling algorithms with underlying suffix-tree structure. This transformation enables controlled experiments and straightforward analysis of the semantics of the component features of PPM and PPM* and of their interactions. The resulting suffix-tree model contains the minimal number of nodes and edges required to represent the same information as the original PPM* implementation, as well as PPM.

3.1 The Structure of PPM and PPM*

PPM's key contribution is its probability estimation technique, which is known as "blending." Blending and its generalization are the subject of Chapter 6. This chapter describes how PPM and PPM* define their model structure in general, and then gives a detailed explanation of how to implement them both with a minimal number of states and transitions.

The PPM FSM has an arbitrary global order bound m and is constructed from the input sequence as it is scanned so that it always satisfies the following invariants:

- Every state s in PPM's state set S is associated with a unique finite context by the function **context** : $S \rightarrow A^*$, where A is the finite input alphabet.

- PPM's state set S is grown incrementally so that for every string w such that $|w| \leq m$ and w has occurred at least once in the input sequence, there exists a unique state $s \in S$ such that $\mathbf{context}(s) = w$.
- PPM's transition function $\delta : S \times A \rightarrow S$ is defined incrementally so that $\delta(p, a) = s$ such that

$$|\mathbf{context}(s)| = \max\{l : l = |\mathbf{context}(t)|,\ t \in S, \mathbf{context}(t) \in \mathit{suffixes}(\mathbf{context}(p) \cdot a)\}.$$

The (non-linear space) PPM* FSM is identical except $m = \infty$.

PPM models can be represented as suffix trees whose transition functions are either computed from the conditioning context strings associated with each state or explicitly represented as pointers. The structure of the PPM suffix tree is defined using a suffix relationship $\mathbf{suffix} : S \rightarrow S$ such that $\mathbf{suffix}(s) = p$ iff $b \cdot \mathbf{context}(p) = \mathbf{context}(s)$, for some $b \in A$. The suffix relationship $\mathbf{suffix} : S \rightarrow S$ is well-defined because for every suffix y of every sequence w such that there exists a (unique) $p \in S$ with $\mathbf{context}(p) = w$, there exists a state $s \in S$ with $\mathbf{context}(s) = y$. Note that it is not necessary to represent the suffix relationship explicitly by storing a suffix pointer at each state [Mof90, CTW95]. The conditioning context partition of any PPM variant is induced by the function $\mathbf{context} : S \rightarrow A^*$. That is, in a PPM or PPM* suffix tree, the language of each state s equals

$$C(s) = A^*\mathbf{context}(s) - \bigcup_{t:\mathbf{suffix}(t)=s} A^*\mathbf{context}(t).$$

The on-line construction of a PPM or PPM* suffix tree from an input sequence $a_1 a_2 \cdots a_n$ is straightforward. Initially, let the initial model consist of the single state s_0 with context string $\mathbf{context}(s_0) = \lambda$. At any time i, the set of *excited states* of the suffix tree equals the set $\{s : a_1 a_2 \cdots a_{i-1} \in A^*\mathbf{context}(s)\}$. After the i^{th} input symbol a_i has been processed, but before the next set of excited states have been determined using $a_1 a_2 \cdots a_i$, we know that $\mathbf{context}(s') = a_{i-|\mathbf{context}(s')|} \cdots a_{i-1}$, where s' equals the maximum order excited state. We can add new states to the suffix tree as follows:

while $|\mathbf{context}(s')| < m$ **do**

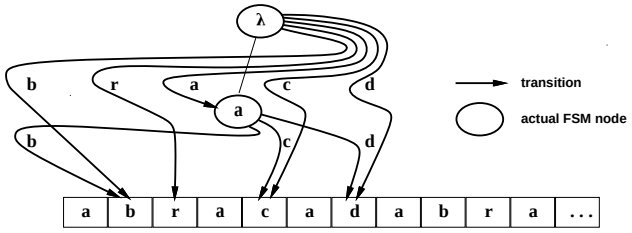

Figure 3.1: **PPM*'s Suffix-Tree FSM for** *abracadabra*.

$t \leftarrow$ *new state*;
suffix$(t) \leftarrow s'$;
context$(t) \leftarrow a_{i-|\mathbf{context}(s')|-1} \cdot \mathbf{context}(s')$;
count$[a_i, t] = 1$;
$s' = t$
end while

After adding new states, the maximum-order excited state, denoted by s', equals the maximum-order newly added state.

Unfortunately, the straightforward procedure described above will construct a suffix tree requiring super-linear space for PPM* models. The authors of PPM* [CTW95] gave a linear-space solution based upon "patricia trees" but it was not clear how this solution related to suffix trees, and it did not use the minimal number of nodes. In the following sections we give our algorithm for the on-line construction of minimal suffix trees for PPM and PPM*.

3.2 Transition Events that are Strings

We can implement PPM* and PPM with a suffix-tree FSM such that: all transitions (edges) correspond to strings, rather than single symbols; all states have at least two outgoing transitions; and the already-processed sequence $a_1 a_2 \cdots a_i$ is stored in an input buffer (shown at the bottoms of Figures 3.1–3.4). By allowing transitions to be labelled as strings, most of the storage normally required to represent states with one in-edge and one out-edge is saved.

The first symbol a of any transition event leaving a given state s is unique for s.

Therefore, we denote transition events as $a|s$. Each transition requires access to the following information:

- its frequency, $\mathbf{count}[a, s]$,
- the input buffer position corresponding to the start of the transition event,
- the length of the transition event, and
- the destination of the transition.

The initial model is constructed as follows. The first $|A|$ positions of the input buffer are initialized to the input alphabet. The string events labeling the transitions exiting state s_{-1} have length one, and for $1 \leq j \leq |A|$, the jth event's input buffer pointer is set to the jth position of the input buffer, which contains the jth symbol of the input alphabet A. The counts of the transitions leaving s_0 are set to define an assumed frequency distribution. We use a uniform distribution, and initialize all the counts to one.

The descriptions and figures below focus on implementing PPM* since all possible PPM models are included in the set of FSMs represented by a single PPM* model. We explain how to place an order bound on the model structure where appropriate.

The suffix-tree FSM (without state s_{-1}) for PPM* and the input sequence *abracadabra* is shown in Figure 3.1. Transitions on finite strings are shown as curved lines terminating with arrows that point to their respective destination states. The fine, straight vertical lines denote the suffix relationship among states. Transitions on unterminated strings, or *unterminated transitions*, are shown pointing to the beginning of their strings in the input buffer. The portion of the input buffer containing the symbols of the input alphabet is not shown.

3.3 Virtual States

Since predictions proceed on a per-symbol basis, we need to compute a probability estimate for each symbol on a given string transition. To do this, we assume that there exists a *virtual state* between every two adjacent symbols on every string transition. The virtual states that correspond to the so-called deterministic states in the original PPM* implementation [CTW95] are shown in Figure 3.2. Since virtual states do not exist in storage, we must deduce the frequency distribution that is conditioned by the conditioning contexts at excited virtual states, in order to use them for probability

estimation.

For each symbol processed, all (actual and virtual) states in the suffix tree whose associated conditioning context partition element contains the current input history sequence become excited. The model's excited states for the sequence *abracadabra* are denoted by numbered arrows, to illustrate their one-to-one correspondence with the states pointed to by the "context list" pictured in the original PPM* illustration [CTW95]. The number on each arrow indicates the Markov order of the indicated state. Note that only the excited virtual states actually exist in memory at any given time. Only virtual states within the global order bound (if there is one) ever become excited, and no virtual states that have been visited after their initial creation exceed the order bound.

The excited virtual states are deduced dynamically for each symbol modeled. The actual transitions on which the virtual states are (virtually) located provides the frequency data that is required for the deduction. A virtual node is dynamically allocated whenever the first symbol of a string transition is crossed, and deallocated when the last symbol of that string transition is processed, or when the input sequence diverges from the transition's string event. Each virtual node records the actual transition on which it resides, the source state of the actual transition, and an offset which records the number of symbols visited since the virtual node was first allocated. The virtual node is also logically associated with a frequency count for its single out event, which equals the frequency of the string transition it resides upon, before that transition's frequency was incremented. The transition's frequency is incremented *after* the first symbol of the string transition is processed, so that the virtual node can be associated with the correct frequency.

Lastly, the currently excited virtual states are logically chained together by the **suffix**() relationship. When a virtual node v that corresponds to a string transition $a|s$ is allocated, if event $a|\mathbf{suffix}(s)$ is a string transition, then that transition has a corresponding excited virtual node v', so $\mathbf{suffix}(v) = v'$. Otherwise, event $a|\mathbf{suffix}(s)$ is a single symbol transition, and $\mathbf{suffix}(v)$ equals the (actual state) destination of transition $a|\mathbf{suffix}(s)$.

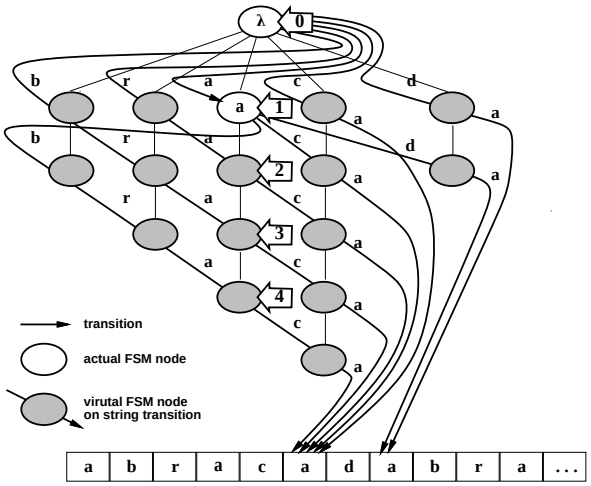

Figure 3.2: **PPM*'s Suffix-Tree FSM for** *abracadabra*, **with Virtual States.**

3.4 Transition Splitting

The PPM* suffix-tree FSM adapts its structure whenever any of the excited states fail to recognize the current symbol. When an excited *actual* state does not recognize the current symbol, a new, unterminated transition is added. The new transition's string is indicated by a pointer from the transition to the current position in the input buffer. When an excited *virtual* state fails to recognize the current symbol, the transition on which the virtual state lies is "split" at that virtual state by replacing the virtual state with an actual state. The new actual node receives two outgoing transitions: the first corresponds to the branch point in the split transition, and the second corresponds exactly to the above case where an excited actual state fails to recognize the current symbol. When all excited states recognize the current symbol, the model structure does not change.

Transition splitting is illustrated in Figure 3.3 which shows how the model evolves when *abracadabra* is followed by a *d*. If the next symbol had been a *c* instead, the FSM would have remained unchanged and the five virtual nodes entered by *c* would have been the next to become excited. Note that virtual nodes for unterminated transitions have no defined out event until the actual transition is being traversed a second time. Thus, the set of excited virtual states will never include virtual states on novel transitions.

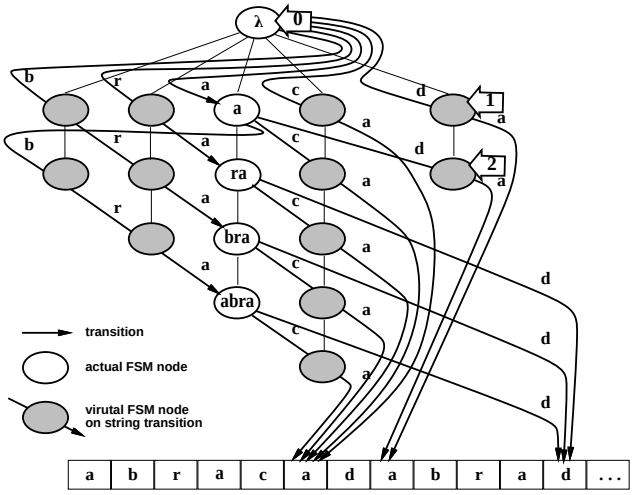

Figure 3.3: **Novel Events and Transition-Splitting in Suffix-Tree PPM***

During transition splitting, if there exists a virtual state whose order exceeds the order bound, the transition that enters that virtual state will be redirected to

1. the maximum-order excited actual state, if its order equals the order bound, or
2. the actual state that replaces the virtual state whose order equals the order bound.

With or without an order bound, splitting must be performed from the bottom up, towards the root, lest subtle errors in transition redirection occur.

3.5 Model Invariants

A single split operation increases the actual suffix-tree FSM by v nodes and v edges, where v is the number of excited virtual nodes within the order-bounded frontier at the time the split was performed. The actual FSM for *abracadabrad* is shown in Figure 3.4. Note how the unterminated transitions leaving the order-zero root state on b and r, and leaving the order-one state labeled with context a on symbol b, all become truncated to finite-length transitions by the splitting operation. Any transition that has been fully traversed twice will have finite length, and will therefore terminate at an actual state.

Incidentally, the lack of branching in the suffix trees shown in Figures 3.1–3.4 is merely an artifact of the example string—each node in an FSMX suffix-tree model is capable of having up to $|A|$ children, one for each possible single-symbol extension aw

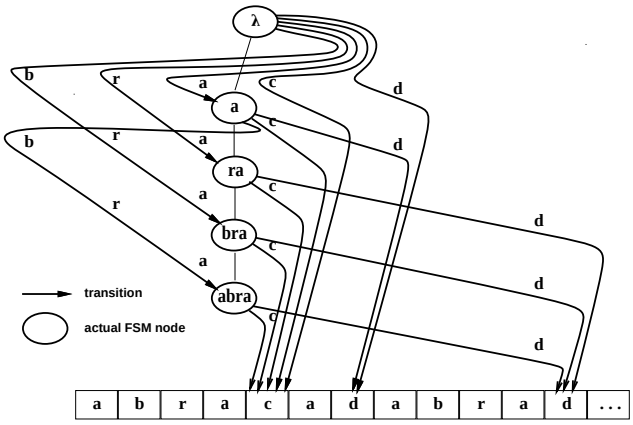

Figure 3.4: **PPM*'s Suffix-Tree FSM for** *abracadabrad*.

of its context w. In general, the suffix-tree FSM presented here satisfies the following invariants, given order bound M (for PPM*, $M = \infty$):

- Every actual node has at least two outgoing transitions.
- For every string w such that $|w| \leq M$ that has been seen at least twice in the past, and always followed by the same symbol, the model will have a virtual node with context string w.
- For every string w such that $|w| \leq M$ that has been seen at least twice in the past, but which has been followed by more than one distinct symbol, the model will have an actual node with context string w.
- Every string such that $|w| \leq M$ that has been seen only once in the past is represented by a virtual node on an unterminated transition.
- No actual nodes, or virtual nodes that have been visited twice will have Markov order exceeding the order bound.

Both PPM* model structures, ours and the original in [CTW95], represent every subsequence of the already-scanned portion of the input. The original PPM and this section's space-saving implementation of it represent every subsequence of the processed input that is no longer than the order bound M. The new model is minimal because the deletion of any actual state or of any transition would cause it to represent less information about the input sequence.

3.6 Space Requirements

Excluding virtual nodes, our suffix-tree FSM has size linear in n, the length of the input sequence. To see why this is true, consider the "forward tree" that is induced by the (actual) suffix-tree FSM, and which corresponds to the original PPM* implementation [CTW95]:

- the forward-child relationship is equal to the next-state relationship,
- the set of nodes include the actual nodes in the FSM plus an added node for each (string-labeled) pointer into the history buffer,
- the leaf nodes of the forward tree are the added nodes, and
- the internal nodes correspond to the actual nodes of the suffix tree.

Each leaf has a unique context: the concatenation of the context of the source of the leaf's history buffer pointer plus the string labeling that pointer. Every leaf node in the forward tree therefore corresponds to two positions in the input history buffer. The end position is given by the pointer into the history buffer, while the beginning position is given by subtracting the length of the node's context from the end position. The beginning position of the sub-buffer pointed to by each forward-tree leaf is unique [CTW95]. Therefore, there are no more forward-tree leaves than there are input symbols. Since all internal nodes have at least two forward-children (actual nodes have at least two string-labeled outgoing transitions), there are no more internal nodes (or equivalently, actual nodes), than there are input symbols. Also, since the arity of the forward tree is at least two, and our suffix-tree implementation does away with all forward-tree leaf nodes, it requires fewer than half of the nodes that are used by the original forward-tree implementation.

Assuming the suffix pointers are replaced with the "context-list" mechanism of the original PPM* implementation, our suffix-tree implementation reduces the space requirements of the original implementation by more than half. How the model parameters are represented is independent of structure.

3.7 Summary

This chapter transformed PPM* to a suffix-tree implementation that requires half the space of PPM*'s original linear-space implementation. An order bound option enables

our implementation to be used to implement PPM with even greater space savings, especially for higher order PPM models. With our implementation, PPM models of arbitrary order can be used on most personal computers, whereas a practical order bound of about 8 existed for standard PPM implementations on modern workstations. The next chapter transforms another popular modeling algorithm, DMC, to a suffix-tree representation. A formal treatment is required to identify the conditioning-context partition of DMC.

Chapter 4

A (SUFFIX-TREE) SEMANTICS FOR DMC

Dynamic Markov Compression (DMC) is one of the simplest and best-performing on-line stochastic data models from the literature that may be combined with arithmetic coding to perform lossless data compression. However, DMC has not been used in practice because of its rapid, unbounded model growth; a problem that has stubbornly resisted solution by means known to work well on related techniques. Here, a rigorous analysis of DMC's FSM identifies the hidden, abstract features that separate DMC from the other techniques, and provides insight required for improving DMC's resource tradeoffs.

We prove a formal semantics for the states of any DMC model, regardless of growth heuristic, in terms of a partition on the set of possible conditioning contexts, described as a subclass of regular languages. Our characterization proves that DMC's ergodic, unifilar, finite-order Markov models do not belong to the class of FSMX sources, which are frequently described in the literature as *properly containing* the class of (finite-order) Markov models. In fact, DMC models are more powerful than FSMX models, for DMC models can recognize languages that any FSMX model can, but the converse is not true.

The characterization enables the first meaningful comparison of DMC models with the many popular FSMX models in the literature, in terms of model structure and *a priori* statistical assumptions. Thus, this work is preliminary to a taxonomical and empirical study that shows how DMC's unique abstract features can be combined with the best features from other influential algorithms to produce truly novel modeling techniques with superior performance.

4.1 Introduction

The popular Dynamic Markov Compression Algorithm (DMC) [CH87] is a member of the family of data compression algorithms that combine an on-line stochastic data model with an arithmetic coder. DMC's distinguishing feature is an elegant but *ad*

hoc modeling technique that provides state-of-the-art compression performance and matchless conceptual simplicity. In practice, however, the cost of DMC's simplicity and performance is often outrageous memory consumption. Several known attempts at reducing DMC's unwieldy model growth (e.g., [Ton93, Yu93]) have rendered DMC's compression performance uncompetitive.

One reason why DMC's model growth problem has resisted solution is that the algorithm is poorly understood. DMC is the only published stochastic data model for which a characterization of its states, in terms of conditioning contexts, is unknown. Up until now, all that was certain about DMC was that a finite-context characterization exists, which was proved in [BM89] using a finiteness argument.

Here, we present and prove the first finite-context characterization of the states of DMC's data model. The impact of our characterization is threefold.

1. It proves that although DMC constructs a unifilar finite-order Markov FSM, DMC states cannot be uniquely characterized by single conditioning contexts, and therefore DMC models do not belong to the class of FSMX automata [Ris86a], which purportedly contain *all* finite order Markovian FSMs.

2. It illuminates principled solutions for curbing counterproductive model growth.

3. It provides a sufficiently general on-line Markov model that can be parameterized to exactly emulate many other influential algorithms from the literature, including many popular FSMX models [CTW95, CW84b, Fur91, RC89, Ris83, Ris86a, WLZ92]. This allows

   (a) precise identification and comparison of the features that distinguish the structure and *a priori* statistical assumptions of different algorithms (such as "state selection," "blending," "update exclusion," etc.), and

   (b) experiments that evaluate the general effectiveness of specific model features by controlling the confounding factors induced by the myriad implementation decisions underlying any empirical evaluation.

   Indeed, this work is preliminary to such a taxonomy and controlled empirical study.

Our analysis reveals that the DMC model, with or without its counterproductive portions, offers abstract structural features not found in other models. Ironically, the space-hungry DMC algorithm actually has a greater potential for economical model representation than its FSMX counterparts have. Once identified, DMC's distinguishing features combine easily with the best features from other techniques. These

combinations have the potential for achieving very competitive compression/memory tradeoffs.

4.2 The DMC Automaton

DMC constructs a series of finite-state machines (FSMs) $M = \{M_0, M_1, \ldots, M_m\}$, where for $0 \leq k \leq m$, $M_k = (S_k, A, \mu_k, s_0)$ such that

s_0 is the initial state;

S_k is the finite set of states, defined recursively as

$$\begin{aligned} S_k &= \{s_0\} \text{ if } k = 0, \\ &= S_{k-1} \cup \{s_k\}, \text{ otherwise;} \end{aligned}$$

A is the finite (input) alphabet; and

$\mu_k : S_k \times A \rightarrow S_k$ is the transition function, which extends to $\mu_k : S_k \times A^* \rightarrow S_k$ in the traditional fashion.[1]

As each symbol in a given input sequence is scanned, DMC applies a refinement eligibility criterion, $\mathcal{E} : M \times A^* \rightarrow \{T, F\}$, to the current FSM, M_k, to decide if it should be extended into M_{k+1}.

The FSMs $M_k, \forall k \geq 0$ are defined recursively. At the start, $\mu_0(s_0, a) = s_0, \forall a \in A$. That is, the initial model consists of the single state, s_0 and a reflexive transition for each symbol in the input alphabet.[2] During processing of an input sequence, whenever $\mathcal{E}(M_k, wa) = T$, a new machine M_{k+1} is constructed from M_k as follows. Let $s_p = \mu_k(s_0, w)$ and $s_t = \mu_k(s_p, a)$. Then, we define

$$\mu_{k+1}(s_p, a) = s_{k+1} \tag{4.1}$$

$$\mu_{k+1}(s_p, b) = \mu_k(s_p, b), \forall b \in A - \{a\} \tag{4.2}$$

$$\mu_{k+1}(s_i, b) = \mu_k(s_i, b), \forall b \in A, \forall i \leq k, i \neq p \tag{4.3}$$

$$\mu_{k+1}(s_{k+1}, b) = \mu_{k+1}(s_t, b), \forall b \in A. \tag{4.4}$$

The process of extending the current FSM M_k to M_{k+1}, called "cloning," is depicted in Figure 4.1. Cloning [CH87] simply redirects the current transition, $\mu_k(s_p, a)$, to a newly added state s_{k+1}. Cloning is performed whenever $\mathcal{E}(M_k, wa) = T$, where

[1] Throughout, we shall denote the concatenation of strings w and y as wy or as $w \cdot y$.

[2] Other possible initial models are given in [CH87].

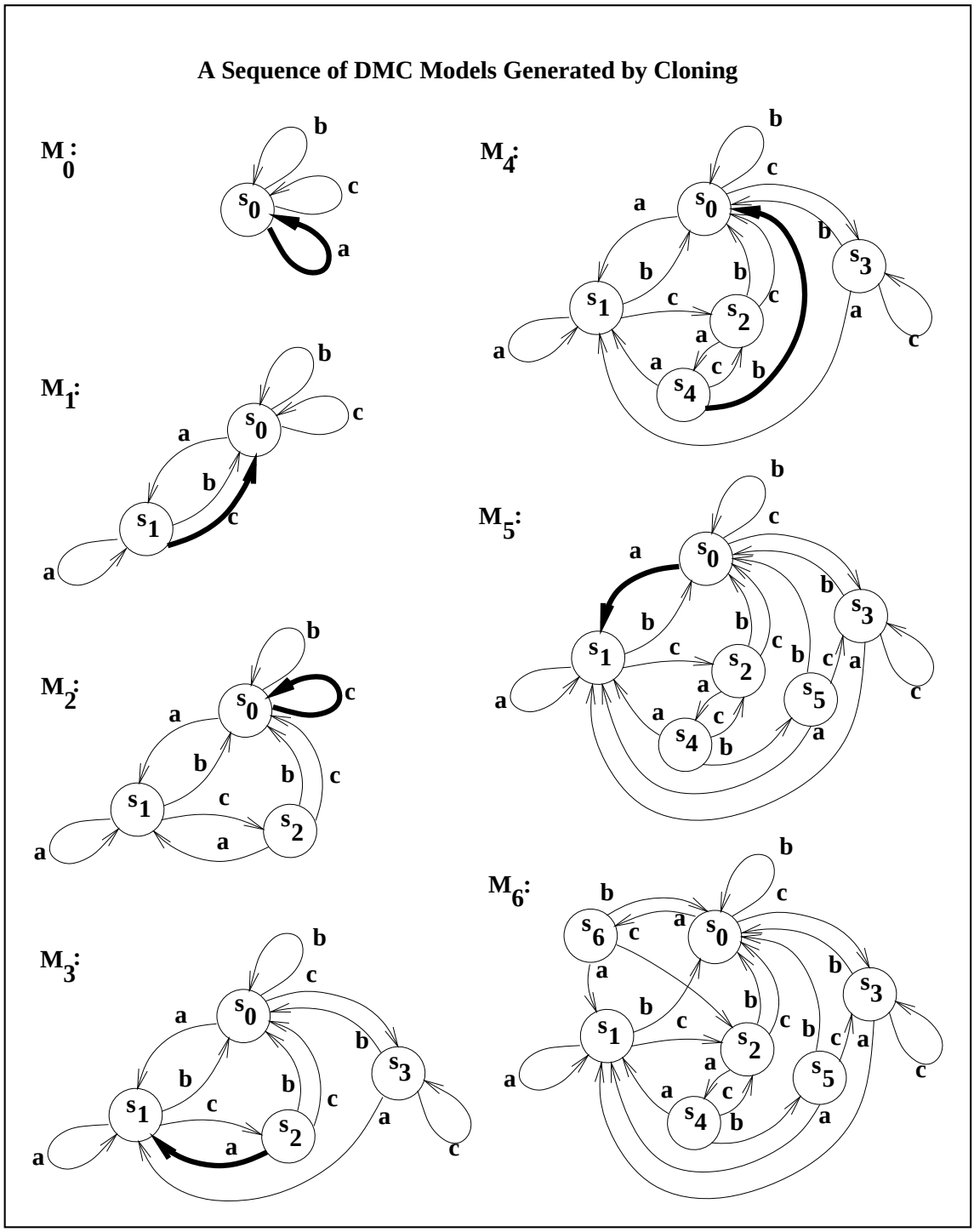

Figure 4.1: **DMC's Finite-State Data Model.** DMC's finite-state data model is created incrementally by cloning the destination of the current transition, if it is determined to be eligible. The **bold** (presumably eligible) transition in each model M_k is redirected to a newly added state s_{k+1} to form model M_{k+1}. The transitions leaving s_{k+1} are copied from the **bold** transition's former destination, after the **bold** transition is redirected.

w is the already scanned portion of the input sequence. The newly added state is a copy of the transition's destination such that the transitions leaving the new state are copied from the transitions leaving the original state. The number of models in M, and the number of states in the final FSM, M_m, equal $m+1$. The number of proper prefixes wa of the given input sequence for which the criterion $\mathcal{E}(M_k, wa)$ holds determines the value of m.

The eligibility criterion given in [CH87] is based upon the popularity of the current transition relative to the popularity of its destination. That is, $\mathcal{E}()$ is equivalent to the well-formed formula

$$t_1 \leq |\{y : y \in \text{ prefixes}(w), \mu_k(s_0, y) = \mu_k(s_0, w), \mu_k(s_0, ya) = \mu_k(s_0, wa)\}| \text{ and}$$
$$t_2 \leq |\{y : y \in \text{ prefixes}(w), \mu_k(s_0, y) \neq \mu_k(s_0, w), \mu_k(s_0, ya) = \mu_k(s_0, wa)\}|,$$

where thresholds t_1 and t_2 are algorithm input parameters. This particular definition of $\mathcal{E}()$ causes DMC to construct a stochastic model that cannot converge to any finite stochastic source that could be assumed to have emitted the sequence. The following analysis of DMC's model structure does not depend upon any particular definition of $\mathcal{E}()$.

4.3 Observable Structure in DMC

4.3.1 Definitions

The following definitions formulate the intuition and axioms of our analysis. They are illustrated in Figure 4.2.

$$\begin{aligned}
\mathbf{prefix}(s_i) &= s_0,\ i = 0 \\
&= s_p : \mu_{i-1}(s_p, a) \neq \mu_i(s_p, a) = s_i, p < i, a \in A,\ i > 0 \\
\mathbf{symbol}(s_i) &= \lambda,\ i = 0,\ \text{where } \lambda \text{ denotes the empty string} \\
&= a : \mu_i(\mathbf{prefix}(s_i), a) = s_i,\ a \in A,\ i > 0 \\
\mathbf{context}(s_i) &= \lambda,\ i = 0 \\
&= \mathbf{context}(\mathbf{prefix}(s_i))\mathbf{symbol}(s_i),\ i > 0 \\
\mathbf{suffix}(s_i) &= s_0,\ i = 0 \\
&= \mu_{i-1}(\mathbf{prefix}(s_i), \mathbf{symbol}(s_i)),\ i > 0 \\
\mathbf{extns}_k(s_i) &= \{s_d : \mathbf{suffix}(s_d) = s_i,\ d \leq k\} \\
\mathbf{extns}^*_k(s_i) &= \mathbf{extns}_k(s_i) \cup \textstyle\bigcup_{s_d \in \mathbf{extns}_k(s_i)} \mathbf{extns}^*_k(s_d)
\end{aligned}$$

The function **prefix** : $S \rightarrow S$ is used with the function **symbol** : $S \rightarrow A$ to recursively map states to finite strings, or contexts. The state **prefix**(s_i) is the source of the transition that was redirected to s_i when s_i was added to the model. The character **symbol**(s_i) labels the transition that was originally redirected to state s_i, and any subsequently added transitions into s_i. We shall prove that any string that brings M_k into state s_i ends in the finite string **context**(s_i), for $k \geq i$.

The function **suffix** : $S \rightarrow S$ organizes the states of M_k into a tree, acting as a parent pointer. The state **suffix**(s_i) is the former destination of the transition that was redirected to state s_i when s_i was created. In the tree induced by M_k, the state s_i is the parent of the states in $\mathbf{extns}_k(s_i)$. Equivalently, the set of states $\mathbf{extns}_k(s_i)$ equals the children of state s_i. The name "**extns**" is used instead of "children" because the relationships among the conditioning contexts associated with each state, rather than their positions in the tree, are the primary points of interest. The closure of the set of children of state s_i equals the set of descendants of s_i, and is denoted $\mathbf{extns}_k^*(s_i)$.

Additionally, two observations from [BM89] are used repeatedly in the proofs of the lemmas and theorems to follow, and they can be restated using the **suffix** function above. The observations basically point out that the only states whose input transitions are affected when s_{k+1} is added are **suffix**(s_{k+1}) and s_{k+1}.

Observation 4.3.1

$$\begin{aligned} &\forall i \leq k, w \in A^*, \\ &\mu_{k+1}(s_i, w) \neq s_{k+1} \quad \Rightarrow \quad \mu_{k+1}(s_i, w) = \mu_k(s_i, w). \end{aligned}$$

Observation 4.3.2

$$\begin{aligned} &\forall i \leq k+1, w \in A^*, \\ &\mu_k(s_i, w) \neq \mu_{k+1}(s_i, w) \quad \Rightarrow \quad \mu_k(s_i, w) = \mathbf{suffix}(s_{k+1}) \textit{ and} \\ &\qquad\qquad\qquad\qquad\qquad\quad \mu_{k+1}(s_i, w) = s_{k+1}. \end{aligned}$$

4.3.2 Contexts of DMC States

Although no semantics have been assigned to DMC states in the related literature, the evocative names of the above functions do imply accurate semantics. That is, the following properties hold for all s_i:

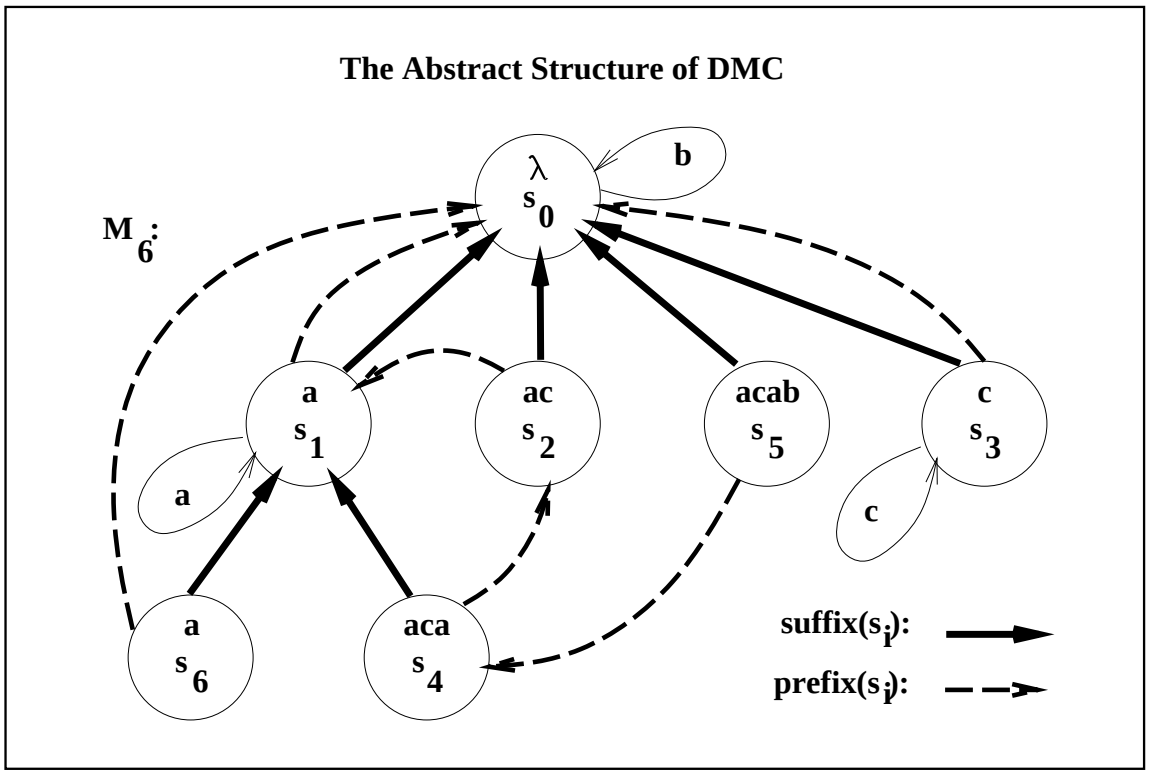

Figure 4.2: **Observable Structure in DMC Models.** For any state s_i, **suffix**(s_i) is the original destination of the transition that was redirected to s_i when s_i was created; **prefix**(s_i) is the source of the transition which was redirected to s_i, when s_i was added to the model; and **symbol**(s_i) labels the transition that was originally redirected to s_i, and any subsequently added transitions into s_i. The context of s_i, **context**(s_i), labels each state.

The non-reflexive transitions of model M_6, pictured in Figure 4.1, are omitted. However, the reflexive transitions of M_6 are included here to illustrate the consistent substructures they define in the DMC model. There are always $|A|$ reflexive transitions in the model. (Here $A = \{a, b, c\}$). When a reflexive transition is redirected by cloning, the newly added state will have a reflexive transition with the same symbol. For any state s_i with a reflexive transition (it can only have one, if $i \neq 0$), **context**(s_i) will be a sequence consisting of a finite number of occurrences of the symbol on the reflexive edge, which equals the state's **symbol**. For example, if the reflexive edge labeled c is redirected to a new state s_7, **prefix**(s_7) will equal s_3 and **context**(s_7) will equal cc.

The state s_6 is an example of a state that has been created by redirecting a *prefix transition*. That is, the original edge which was redirected to point to s_1 was redirected again when s_6 was added to the model. Both s_6 and s_1 therefore have identical **context**s.

- [Prefixes] **context**(**prefix**(s_i)) is a prefix of **context**(s_i),
- [Suffixes] **context**(**suffix**(s_i)) is a suffix of **context**(s_i),
- [Extensions] $s_d \in \mathbf{extns}^*_k(s_i)$ implies that **context**(s_d) extends **context**(s_i) on the left by zero or more symbols.

Thus, the **context**() of a state in a DMC model is analogous to the context of a state in an FSMX model [Ris86a]. However, there are differences. For one thing, **context**(**prefix**(s_i)) is exactly one symbol shorter than **context**(s_i), while **context**(s_i) may be arbitrarily longer than **context**(**suffix**(s_i)). That is, DMC builds a context model with variable-length *minimal extensions*[3] of a context. Furthermore, if the eligibility criterion $\mathcal{E}()$ allows transitions to be redirected more than once, as DMC's does, it is possible that **context**(**suffix**(s_i)) = **context**(s_i).

4.3.3 Reflexive Edges in DMC

Reflexive edges appear only under certain circumstances in DMC models, and any M_k will have reflexive edges if and only if the initial state s_0 does. Intuitively, Lemma 4.3.1 states that any other reflexive edges can only be created by redirecting reflexive edges. When a reflexive edge is redirected, the **prefix** of the new destination state equals that state's **suffix**. Over time, a reflexive edge will either stay the same or will point down into its source state's extensions (subtree). Conversely, any state with a self-loop, or an edge that points down into its subtree, was initially added to the model as as the new destination of a redirected reflexive edge.

Only the base case of the following lemma, that is, a description of reflexive edges leaving any novel state s_k in model M_k, is required for the proof of Theorem 4.4.1, the complete characterization of DMC's structure. However, by proving the lemma for all states i in any model M_k, we can describe the behavior of all reflexive edges in DMC models over time. Incidentally, when $i = 0$, the left hand side of the equivalence holds trivially. That is, all edges leaving the root state s_0 enter either s_0 or a node in its subtree.

Lemma 4.3.1 (Reflexive Edge Characterization)

$$\forall k \geq 1,\ \forall i, 1 \leq i \leq k,\ \forall a \in A,$$

[3] A *minimal extension* is the $y \in A^*$ *s.t.* **context**(s_i) = $y \cdot$ **context**(**suffix**(s_i)).

$$\mu_k(s_i, a) \in \{s_i\} \cup \mathbf{extns}^*_k(s_i) \;\Leftrightarrow\; \mathbf{suffix}(s_i) = \mathbf{prefix}(s_i) \textit{ and } \mu_{i-1}(\mathbf{suffix}(s_i), a) = \mathbf{suffix}(s_i)$$

The proof proceeds by induction on $k - i$. To prove the induction base, where $k = i$ and

$$\mu_k(s_k, a) \in \{s_k\} \cup \mathbf{extns}^*_k(s_k) \;\Leftrightarrow\; \mathbf{suffix}(s_k) = \mathbf{prefix}(s_k) \text{ and } \mu_{k-1}(\mathbf{suffix}(s_k), a) = \mathbf{suffix}(s_k),$$

we first recall from the definition of M_k that

$$\mu_{k-1}(\mathbf{prefix}(s_k), a) = \mathbf{suffix}(s_k), \tag{4.5}$$

$$\mu_k(\mathbf{prefix}(s_k), a) = s_k, \tag{4.6}$$

and

$$\forall b \in A,\ \mu_k(s_k, b) = \mu_k(\mathbf{suffix}(s_k), b). \tag{4.7}$$

The proof is straightforward:

$\mu_{k-1}(\mathbf{suffix}(s_k), a) = \mathbf{suffix}(s_k)$ and $\mathbf{prefix}(s_k) = \mathbf{suffix}(s_k)$

$\Rightarrow$ $\mu_{k-1}(\mathbf{suffix}(s_k), a) \neq \mu_k(\mathbf{suffix}(s_k), a)$ and
$\mu_{k-1}(\mathbf{prefix}(s_k), a) \neq \mu_k(\mathbf{prefix}(s_k), a)$ — by (4.5), (4.6), subst.

$\Rightarrow$ $\mu_k(\mathbf{suffix}(s_k), a) = s_k$ — by Observation 4.3.2

$\Rightarrow$ $\mu_k(s_k, a) = s_k$ — by (4.7), subst.

$\Rightarrow$ $\mu_k(s_k, a) \in \{s_k\} \cup \mathbf{extns}^*_k(s_k)$ — $\mathbf{extns}^*_k(s_k) = \{\}$.

$\mu_k(s_k, a) \in \{s_k\} \cup \mathbf{extns}^*_k(s_k)$

$\Rightarrow$ $\mu_k(s_k, a) = s_k$ — $\mathbf{extns}^*_k(s_k) = \{\}$.

$\Rightarrow$ $\mu_k(\mathbf{suffix}(s_k), a) = s_k$ — by (4.7), subst.

$\Rightarrow$ $\mu_{k-1}(\mathbf{suffix}(s_k, a) \neq \mu_k(\mathbf{suffix}(s_k, a)$ — since$s_k \notin S_{k-1}$

$\Rightarrow$ $\mu_{k-1}(\mathbf{suffix}(s_k, a) = \mathbf{suffix}(s_k)$ — by Observation 4.3.2

$\Rightarrow$ $\mu_{k-1}(\mathbf{suffix}(s_k), a) = \mu_{k-1}(\mathbf{prefix}(s_k), a)$, and
$\mu_k(\mathbf{suffix}(s_k), a) = \mu_k(\mathbf{prefix}(s_k), a)$ — by (4.5) and (4.6), subst.

$\Rightarrow$ $\mu_{k-1}(\mathbf{prefix}(s_k, a) \neq \mu_k(\mathbf{prefix}(s_k, a)$ — subst

$\Rightarrow$ $\mathbf{prefix}(s_k) = \mathbf{suffix}(s_k)$ — since only one transition in M_{k-1} changed to form M_k.

$\Rightarrow$ $\mu_{k-1}(\mathbf{suffix}(s_k), a) = \mathbf{suffix}(s_k)$ and
$\mathbf{prefix}(s_k) = \mathbf{suffix}(s_k)$.

For the induction step, assume the statement holds for a given state $s_i, i > 0$, in model M_k and consider the same state s_i in model M_{k+1}. Note that $s_i \neq s_{k+1}$ because $s_{k+1} \notin S_k$.

Assume $\mu_{k+1}(s_i, a) \in \mathbf{extns}^*_{k+1}(s_i) \cup \{s_i\}$. Then, there are two mutually exclusive cases to consider: $\mu_{k+1}(s_i, a)/neqs_{k+1}$ and $\mu_{k+1}(s_i, a) = s_{k+1}$.

$\mu_{k+1}(s_i, a) \neq s_{k+1}$
$\Rightarrow$ $\mu_{k+1}(s_i, a) = \mu_k(s_i, a)$ by Observation 4.3.1
$\Rightarrow$ $\mathbf{suffix}(s_i) = \mathbf{prefix}(s_i)$ and
$\mu_{i-1}(\mathbf{suffix}(s_i), a) = \mathbf{suffix}(s_i)$ by ind. hyp., subst.

$\mu_{k+1}(s_i, a) = s_{k+1}$
$\Rightarrow$ $\mu_{k+1}(s_i, a) \neq \mu_k(s_i, a)$ $s_{k+1} \notin S_k$
$\Rightarrow$ $\mu_k(s_i, a) = \mathbf{suffix}(s_{k+1})$ by Obs. 4.3.2
$\Rightarrow$ $\mu_k(s_i, a) \in \mathbf{extns}^*_{k+1}(s_i) - \{s_{k+1}\}$ $\mu_k(s_i, a) \in \mathbf{extns}^*_{k+1}(s_i)$ and $\mu_k(s_i, a) \neq s_{k+1}$
$\Rightarrow$ $\mu_k(s_i, a) \in \mathbf{extns}^*_k(s_i)$ $\mathbf{extns}^*_k(s_i) = \mathbf{extns}^*_{k+1}(s_i) - \{s_{k+1}\}$
$\Rightarrow$ $\mathbf{suffix}(s_i) = \mathbf{prefix}(s_i)$ and
$\mu_{i-1}(\mathbf{suffix}(s_i), a) = \mathbf{suffix}(s_i)$ by ind. hyp.

Conversely, assume $\mathbf{suffix}(s_i) = \mathbf{prefix}(s_i)$ and $\mu_{i-1}(\mathbf{suffix}(s_i), a) = \mathbf{suffix}(s_i)$. We know by the induction hypothesis that $\mu_k(s_i, a) \in \mathbf{extns}^*_k(s_i) \cup \{s_i\}$. Either $\mu_{k+1}(s_i, a) = \mu_k(s_i, a)$, and so $\mu_{k+1}(s_i, a) \in \mathbf{extns}^*_{k+1}(s_i) \cup \{s_i\}$ because $\mathbf{extns}^*_k(s_i) \subseteq \mathbf{extns}^*_{k+1}(s_i)$; or $\mu_{k+1}(s_i, a) \neq \mu_k(s_i, a)$. In the latter case,

$\mu_{k+1}(s_i, a) \neq \mu_k(s_i, a)$
$\Rightarrow$ $\mu_k(s_i, a) = \mathbf{suffix}(s_{k+1})$ by Observation 4.3.2
$\Rightarrow$ $\mathbf{suffix}(s_{k+1}) \in \mathbf{extns}^*_k(s_i) \cup \{s_i\}$ by ind. hyp., subst.
$\Rightarrow$ $s_{k+1} \in \mathbf{extns}^*_{k+1}(s_i)$ by def. $\mathbf{extns}^*()\Box$

Lemma 4.3.1 implies the following regularities for DMC models, assuming the initial model given earlier:

- For each $b \in A$ there exists exactly one reflexive edge labeled b.
- No state other than s_0 may have more than one reflexive edge.

- And, if a state s_i has a reflexive edge labeled a, then $\mathbf{context}(s_i) = a^h$, where h is the length of the path of **prefix**() pointers from s_i to s_0.

Lastly, the following technical corollary to Lemma 4.3.1 will be required to prove the upcoming DMC characterization. If a string brings a model M_{k+1} into the new state s_{k+1} and it also brought the preceding model M_k to the **prefix** state of the new state, then the new state must have a reflexive out-edge, labeled with the state's **symbol**.

Corollary 4.3.1 (to Reflexive Edge Characterization)

$$\begin{aligned}
&\forall k, \forall j \leq k, \forall w \in A^*, \\
&\mu_k(s_j, w) = \mathbf{prefix}(s_{k+1}) \text{and} \mu_{k+1}(s_j, w) = s_{k+1} \\
&\quad \Rightarrow \quad \mu_{k+1}(s_{k+1}, \mathbf{symbol}(s_{k+1})) = s_{k+1}.
\end{aligned}$$

Proof:

$$\begin{array}{lll}
\mu_k(s_j, w) = \mathbf{prefix}(s_{k+1}) \text{ and } \mu_{k+1}(s_j, w) = s_{k+1} & & \\
\quad \Rightarrow \quad \mu_k(s_j, w) \neq \mu_{k+1}(s_j, w) & & \mathbf{prefix}(s_{k+1}) \neq s_{k+1} \\
\quad \Rightarrow \quad \mu_k(s_j, w) = \mathbf{suffix}(s_{k+1}) & & \text{by Observation 4.3.2} \\
\quad \Rightarrow \quad \mathbf{prefix}(s_{k+1}) = \mathbf{suffix}(s_{k+1}) & & \text{subst.} \\
\quad \Rightarrow \quad \mu_k(\mathbf{prefix}(s_{k+1}), \mathbf{symbol}(s_{k+1})) = \mathbf{suffix}(s_{k+1}) & & \text{by def. } \mathbf{suffix}(s_{k+1}) \\
\quad \Rightarrow \quad \mu_{k+1}(\mathbf{prefix}(s_{k+1}), \mathbf{symbol}(s_{k+1})) = s_{k+1} & & \text{by def. } \mu_{k+1}() \\
\quad \Rightarrow \quad \mu_k(\mathbf{suffix}(s_{k+1}), \mathbf{symbol}(s_{k+1})) = \mathbf{suffix}(s_{k+1}) & & \text{subst.} \\
\quad \Rightarrow \quad \mu_{k+1}(s_{k+1}, \mathbf{symbol}(s_{k+1})) = \{s_{k+1}\} \cup \mathbf{extns}^*_{k+1}(s_{k+1}) & & \text{Lemma 4.3.1, contrap.} \\
\quad \Rightarrow \quad \mu_{k+1}(s_{k+1}, \mathbf{symbol}(s_{k+1})) = s_{k+1} & & \mathbf{extns}^*_{k+1}(s_{k+1}) = \{\} \Box
\end{array}$$

4.4 A Finite-Order Characterization of DMC States

A finite-state stochastic model, such as DMC's, is traditionally defined in terms of its *structure* and *parameters*. The language of an individual FSM state is the set of strings which put the FSM into the given state, by following the successive transitions labelled with each symbol of the string left-to-right, starting at the FSM's initial state. The set consisting of each language of each state in a stochastic model forms a partition on the set of possible input sequences to the model, that is, a *context partition*. The *structure* of a stochastic model is defined by its context partition.

Thus each state in the model is associated with a single class of strings, and the model is used to successively classify each progressively longer prefix of an entire input sequence. To use such a model to estimate probabilities of a sequence, or to predict or generate such a sequence, each state in the model must also be associated with a (usually empirical) probability measure on the symbols that may immediately follow any string belonging to the state's class. DMC's solution to this *parameterization* problem is described in Section 6.5.

DMC is unique in the data compression literature, in that its model was not originally defined to represent a given context partition. The goal of this analysis is to identify the hitherto unknown context partition of any DMC model, so that aspects of the DMC technique may be meaningfully compared with features of other techniques in the literature. DMC's context partition is given below by the function $C_k : S \rightarrow 2^{A^*}$, which, as we prove, describes the language of any state in a DMC model M_k.

Theorem 4.4.1 (Characterization of DMC Structure)

$$\begin{aligned} \textit{Let} \quad C_k(s_i) &= \mathbf{L}(s_i) - \bigcup_{s_d \in \mathbf{extns}_k(s_i)} \mathbf{L}(s_d), \qquad (4.8) \\ \textit{where} \ \mathbf{L}(s_i) &= A^*, \ \textit{if} \quad i = 0 \\ &= C_{i-1}(\mathbf{prefix}(s_i)) \cdot \mathbf{symbol}(s_i), \quad \textit{otherwise.} \end{aligned}$$

Then, $\forall k,\ \forall i \leq k,$ *and* $\forall w \in A^*,$

$$w \in C_k(s_i) \quad \Leftrightarrow \quad \mu_k(s_0, w) = s_i. \qquad (4.9)$$

Before we prove that the characterization $C_k()$ is correct, a discussion of some properties of the functions introduced above will provide some insight.

The function $C_k : S_k \rightarrow 2^{A^*}$ maps each state to a set of conditioning contexts. Equation (4.9) implies that the sets of distinct states are disjoint; that is, $C_k()$ relies on the function $\mathbf{L}()$, which is one-to-one regardless of the definition of $\mathcal{E}()$. The regular set $\mathbf{L}(s_i)$ precisely describes the set of strings that bring M_k to state s_i or to any descendant of s_i. Note that although the language $C_k(s_i)$ of a state s_i changes with k (which is monotone increasing), the language of the subtree rooted by a state s_i, that is, $\mathbf{L}(s_i)$, does not (even though the subtree itself may change in structure).

The set $\mathbf{L}(s_i) \subseteq A^*\mathbf{context}(s_i)$, and recursive expansions of the regular expression $\mathbf{L}(s_i)$ quit branching at all states s_p such that $\mathbf{L}(s_p) = A^*\mathbf{context}(s_p)$. Figure 4.3 shows the DMC model substructures relevant to any arbitrary state s_i, and illustrates the recursive expansion of expressions $C_k(s_i)$ and $\mathbf{L}(s_i)$.

The function $\mathbf{L}()$ is well-defined, and would still be well-defined even if we did not require the recursive expansion to continue down to the initial state s_0. This can be accomplished by optimizing the terminating expansion given above:

$$\begin{aligned}\mathbf{L}(s_i) &= A^* \cdot \mathbf{context}(s_i), \text{ if } i = 0 \text{ or } \forall j \leq i, \\ &\quad \mathbf{prefix}(s_j) \neq \mathbf{prefix}(\mathbf{suffix}(s_j)) \text{ and } \mathbf{extns}_{j-1}(\mathbf{prefix}(s_j)) = \{\} \quad (4.10) \\ &= C_{i-1}(\mathbf{prefix}(s_i)) \cdot \mathbf{symbol}(s_i), \text{ otherwise.}\end{aligned}$$

The resulting regular expressions for the languages of each state are the same with either definition, but the terminating criterion used in Equation (4.10) above creates the shallowest recursive expansion. Furthermore, the model condition required by Equation (4.10), the base of the inductive definition, describes the exact requirements for the states of a DMC model to be characterizable by single finite strings. Note that in FSMX models, for all states s_j, $\mathbf{prefix}(s_j) \neq \mathbf{prefix}(\mathbf{suffix}(s_j))$ and $\mathbf{extns}_{j-1}(\mathbf{prefix}(s_j)) = \{\}$. Thus for all states s_i in an FSMX model, $L(s_i) = A^*\mathbf{context}(s_i)$; that is, all states in FSMX models are characterizable by single finite strings.

4.4.1 Correctness Proof of the DMC Characterization

Here we prove Theorem 4.4.1, which states that Equation (4.8) characterizes the partition of conditioning contexts that is induced by the states of any model M_k, by proving that $C_k : S \rightarrow 2^{A^*}$ describes the language of each state in M_k, for all k. The proof proceeds by induction on k, the number of states that have been added to M_0 to create the k progressive refinements to M_0 which result in model M_k.

The induction base is trivial: $\mu_0(s_0, w) = s_0$ and $w \in C_0(s_0) = A^*\lambda$. $C_0(s_0) = A^*\lambda$ because $\mathbf{extns}_0(s_0) = \{\}$ and $\mathbf{L}(s_0) = A^*\lambda$.

For the induction step we assume that

$$\begin{aligned} &\forall h \leq k,\ \forall i \leq h, \quad \text{and } \forall w \in A^*, \\ &\qquad w \in C_h(s_i) \quad \Leftrightarrow \quad \mu_h(s_0, w) = s_i, \end{aligned}$$

and prove that

$$\begin{aligned} &\forall i \leq k+1, \quad \text{and } \forall w \in A^*, \\ &\qquad w \in C_{k+1}(s_i) \quad \Leftrightarrow \quad \mu_{k+1}(s_0, w) = s_i. \end{aligned}$$

Case 1:

When $w = \lambda$, we know that $w \in C_{k+1}(s_i) \Leftrightarrow s_i = s_0 \Leftrightarrow \mu_{k+1}(s_0, w) = s_i$. This equivalence follows from the fact that $s_d \in \mathbf{extns}(s_i) \Rightarrow \mathbf{symbol}(s_d) = \mathbf{symbol}(s_i)$ and the definition of $C_k()$, which expands to form

$$\begin{aligned} C_k(s_0) &= A^* - \bigcup_{s_d \in \mathbf{extns}_k(s_i)} C_{d-1}(\mathbf{prefix}(s_d)) \cdot \mathbf{symbol}(s_d), \text{and} \\ C_k(s_{i\neq 0}) &= \left(C_{i-1}(\mathbf{prefix}(s_i)) - \bigcup_{s_d \in \mathbf{extns}_k(s_i)} C_{d-1}(\mathbf{prefix}(s_d)) \right) \cdot \mathbf{symbol}(s_i). \end{aligned}$$

Thus, $C_k(s_0)$ contains λ and $C_k(s_{i\neq 0})$ only contains strings ending in $\mathbf{symbol}(s_i)$.

Case 2:

When $w \neq \lambda$, we let $w = va$ and consider two cases: $s_i = s_{k+1}$ and $s_i \neq s_{k+1}$. The complete proofs for each case are given below.

Proof that $va \in C_{k+1}(s_{k+1}) \Leftrightarrow \mu_{k+1}(s_0, va) = s_{k+1}$*:*

$va \in C_{k+1}(s_{k+1})$

$\Leftrightarrow$ $va \in \mathbf{L}(s_{k+1}) - \bigcup_{s_d \in \mathbf{extns}_{k+1}(s_{k+1})} \mathbf{L}(s_d)$ by def. $C_{k+1}()$

$\Leftrightarrow$ $va \in \mathbf{L}(s_{k+1})$ $\mathbf{extns}_{k+1}(s_{k+1}) = \{\}$

$\Leftrightarrow$ $va \in C_k(\mathbf{prefix}(s_{k+1})) \cdot \mathbf{symbol}(s_{k+1})$ by def. $\mathbf{L}()$

$\Leftrightarrow$ $v \in C_k(\mathbf{prefix}(s_{k+1}))$ and $a = \mathbf{symbol}(s_{k+1})$ substitution; $s_{k+1} \neq s_0$

$v \in C_k(\mathbf{prefix}(s_{k+1}))$, $a = \mathbf{symbol}(s_{k+1})$

$\Rightarrow$ $\mu_k(s_0, v) = \mathbf{prefix}(s_{k+1})$, by ind. hyp., Case 1

$\Rightarrow$ Case: $\mu_{k+1}(s_0, v) \neq s_{k+1}$

$\Rightarrow$ $\mu_{k+1}(s_0, v) = \mu_k(s_0, v) = \mathbf{prefix}(s_{k+1})$ Obs. 4.3.1

Case: $\mu_{k+1}(s_0, v) = s_{k+1}$

$\Rightarrow$ $\mu_{k+1}(s_{k+1}, \mathbf{symbol}(s_{k+1})) = s_{k+1}$ Corollary 4.3.1

$\Rightarrow$ $\mu_{k+1}(\mu_{k+1}(s_0, v), \mathbf{symbol}(s_{k+1})) = s_{k+1}$

$\Rightarrow$ $\mu_{k+1}(s_0, va) = s_{k+1}$ substitution

$\mu_{k+1}(s_0, va) = s_{k+1}$

$\Rightarrow$ $\mu_{k+1}(\mu_{k+1}(s_0, v), \mathbf{symbol}(s_{k+1})) = s_{k+1}$

$\Rightarrow$ Case: $\mu_{k+1}(s_0, v) \neq s_{k+1}$

$\Rightarrow$ $\mu_{k+1}(s_0, v) = \mu_k(s_0, v) = \mathbf{prefix}(s_{k+1})$; Obs. 4.3.1

Case: $\mu_{k+1}(s_0, v) = s_{k+1}$

$\Rightarrow$ $\mu_{k+1}(s_0, v) \neq \mu_k(s_0, v)$ $s_{k+1} \notin S_k$

$\Rightarrow$ $\mu_k(s_0, v) = \mathbf{suffix}(s_{k+1})$ Obs. 4.3.2

$\Rightarrow$ $\mu_{k+1}(\mu_{k+1}(s_0, v), a) = \mu_{k+1}(s_{k+1}, \mathbf{symbol}(s_{k+1})) = s_{k+1}$ substitution

$\Rightarrow$ $\mu_k(\mathbf{suffix}(s_{k+1}), \mathbf{symbol}(s_{k+1})) = \mathbf{suffix}(s_{k+1})$

and $\mathbf{suffix}(s_{k+1}) = \mathbf{prefix}(s_{k+1})$ Lemma 4.3.1

$\Rightarrow$ $\mu_k(s_0, v) = \mathbf{prefix}(s_{k+1})$ substitution.

$\Rightarrow$ $v \in C_k(\mathbf{prefix}(s_{k+1}))$ and $a = \mathbf{symbol}(s_{k+1})$ by ind. hyp., Case 1$\Box$

Proof that for $i \neq k+1$, $va \in C_{k+1}(s_i) \Leftrightarrow \mu_{k+1}(s_0, va) = s_i$:

Here, the contrapositive of the result proved in Section 4.4.1, above, shall be useful:

$$va \notin C_{k+1}(s_{k+1}) \quad \Leftrightarrow \quad \mu_{k+1}(s_0, va) = s_i \text{ for some } i \neq k+1. \tag{4.11}$$

Furthermore, the fact that the languages of disjoint subtrees are disjoint is required. That is,

Claim 4.4.1

$$va \in C_{k+1}(s_i),\ i \neq k+1,\ \textit{and}\ s_i \neq \mathbf{suffix}(s_{k+1}) \quad \Rightarrow \quad va \notin \mathbf{L}(s_{k+1})$$

Proof: We proceed by a proof by contradiction.

$s_i \neq \mathbf{suffix}(s_{k+1})$ and $va \in C_{k+1}(s_i)$
- $\Rightarrow$ $C_{k+1}(s_i) = C_k(s_i)$ and $va \in C_k(s_i)$ — $\mathbf{extns}_{k+1}(s_i) = \mathbf{extns}_k(s_i)$
- $\Rightarrow$ $\mu_k(s_0, va) = s_i$ — by ind. hyp.

$va \in \mathbf{L}(s_{k+1})$
- $\Rightarrow$ $va \in C_k(\mathbf{prefix}(s_{k+1})) \cdot \mathbf{symbol}(s_{k+1})$ — by def. $\mathbf{L}()$
- $\Rightarrow$ $\mu_k(s_0, v) = \mathbf{prefix}(s_{k+1})$ — by ind. hyp, Case 1.
- $\Rightarrow$ $\mu_k(s_0, va) = \mathbf{suffix}(s_{k+1})$ — by def. $\mathbf{suffix}()(\rightarrow\leftarrow)$

The remainder of the proof of Theorem 4.4.1 follows easily:

$va \in C_{k+1}(s_i), i \neq k+1$

- $\Leftrightarrow$ $va \in \left(\mathbf{L}(s_i) - \bigcup_{s_d \in \mathbf{extns}_{k+1}(s_i)} \mathbf{L}(s_d)\right)$ — by def. $C_{k+1}()$
- $\Leftrightarrow$ $va \in \left(\mathbf{L}(s_i) - \bigcup_{s_d \in \mathbf{extns}_k(s_i)} \mathbf{L}(s_d)\right) - \mathbf{L}(s_{k+1})$ — $s_i \neq \mathbf{suffix}(s_{k+1})$ and Claim 4.4.1 or $s_i = \mathbf{suffix}(s_{k+1})$, therefore $s_{k+1} \in \mathbf{extns}_k(s_i)$
- $\Leftrightarrow$ $va \in \left(\mathbf{L}(s_i) - \bigcup_{s_d \in \mathbf{extns}_k(s_i)} \mathbf{L}(s_d)\right) - \left(\mathbf{L}(s_{k+1}) - \bigcup_{s_d \in \mathbf{extns}_{k+1}(s_{k+1})} \mathbf{L}(s_d)\right)$ — $\mathbf{extns}_{k+1}(s_{k+1}) = \{\}$
- $\Leftrightarrow$ $va \in C_k(s_i) - C_{k+1}(s_{k+1})$ — by def. $C_k()$
- $\Leftrightarrow$ $va \in C_k(s_i)$ and $va \notin C_{k+1}(s_{k+1})$
- $\Leftrightarrow$ $\mu_k(s_0, va) = s_i$ and $\mu_{k+1}(s_0, va) \neq s_{k+1}$ — by ind. hyp. and (4.11)
- $\Leftrightarrow$ $\mu_{k+1}(s_0, va) = s_i$ — $i \neq k+1$, Obs. 4.3.1 $\square$

4.5 DMC vs. Other Stochastic Data Models

There are three important aspects of a stochastic data modeling technique that may distinguish it abstractly from others: the language family of the models constructed, which ultimately determines the limitations of the technique; model structure, which determines how the model organizes the data it gathers; and the statistical assumptions, which determine how the frequency data is updated and combined to compute probability estimates.

These are not orthogonal issues. Usually the frequency data are organized with respect to conditioning contexts, which correspond to partitions on the set of possible strings. Since a model's structure certainly determines the languages its states recognize, a structural point of view is, for the most part, a different (and more intuitive) way of looking at the generative power of the model. But only for the most part: two models that recognize the same languages can have very different structure, and therefore may organize frequency data differently.

4.5.1 Linguistic Power

Given that the partition element that corresponds to an arbitrary DMC model state cannot be described using a single finite suffix, what, generally speaking, does it take to describe it? There is a known family of languages that contains the family of languages recognizable by DMC FSMs more tightly than the class of regular languages: For any DMC model state s, $C_k(s)$ is a *locally testable star-free* regular set. *Star-free* regular languages are regular sets that can be described using a finite number of set concatenations, unions, and complements, [vL90, Chapter 1].

To see that $C_k(s_i)$ is star-free for any s_i, simply replace set subtraction with intersection of the absolute complement of the subtrahend, and A^* with $\overline{\{\}}$. $C_k(s_i)$ is also *locally testable*, which means expressible as finite Boolean combinations of sets of the type FA^*, A^*G, or A^*HA^*, where F, G, H are finite sets. However, this is a loose characterization, for it was shown in [BM89] the languages recognizable by DMC models are contained, possibly properly, in the class of languages expressible as $A^*F \cup G$ (where F and G are finite sets), that is, *finite-order* languages. Below, we give a new proof of that containment, which also shows that the characterization of Theorem 4.4.1, $C_k()$ is *finite-order*, as it should be.

Intuitively, DMC models belong to the class of *finite-order Markov sources* [Ash65], also known as *Finite-Context Automata* [BM89], because only a finite suffix of a given source string is required to determine the state to which the string will carry a given model. This means, for one thing, that DMC FSMs cannot recognize infinitely repeating patterns, a common capability of more general FSMs. Neither, however, can any of the popular FSMX models [CW84b, CTW95, Fur91, RC89, Ris83, Ris86a, WLZ92, Wil91], all of which can determine the current model state by locating the state whose single associated finite context is a maximal suffix of the given input string. On the other hand, while a given state in an FSMX model can be uniquely specified with a single finite string, an arbitrary DMC state cannot. Therefore, even when stripped of its implementation details and Bayesian assumptions for probability estimation, DMC proves to be distinct from the other stochastic models in the literature.

4.5.2 DMC Models Have Finite Order

The regular expression $C_k(s_i)$ may be described as a *finite-order* characterization of the strings which bring M_k to s_i from the initial state s_0, because deciding membership of any string wa in the language of any state s_i requires a finite number of comparisons with a finite number of symbols at the end of the string wa. The concept, *finite-order*, is easily formalized for regular expressions (and therefore for finite-state machines), and leads to a simple proof that DMC constructs finite-order FSMs.

A language L over a finite alphabet A is *finite order* if and only if there exist finite sets of strings F and G such that $L = A^*F \cup G$.

Finite-Order Expressions over A and the languages they denote are defined recursively:

1. A^* is a finite-order expression and $L(A^*) = A^*$.
2. If α and β are finite-order expressions over A and $a \in A$ then $\alpha \cup \beta$, $\alpha \cap \beta$, $\overline{\alpha}$, and $\alpha \cdot a$ are finite-order expressions, where

$$\begin{aligned} L(\alpha \cap \beta) &= L(\alpha) \cap L(\beta) \\ L(\alpha \cup \beta) &= L(\alpha) \cup L(\beta) \\ L(\overline{\alpha}) &= A^* - L(\alpha) \\ L(\alpha \cdot a) &= L(\alpha) \cdot a. \end{aligned}$$

Theorem 4.5.1 *A language L is finite-order iff $L = L(\alpha)$, for some finite-order expression α.*

Proof: To show that any finite order language can be expressed as a finite-order expression, we consider a finite-order language $L = A^*F \cup G$, where $F = \{x_1, \ldots, x_n\}$, and $G = \{y_1, \ldots, y_m\}$, and then express the sets A^*F, and G, as finite-order expressions:

$$\begin{aligned} A^*F &= A^*x_1 \cup A^*x_2 \cup \cdots \cup A^*x_n, \quad \text{and} \\ G &= \{\lambda\}y_1 \cup \{\lambda\}y_2 \cup \cdots \cup \{\lambda\}y_m \\ &= \overline{A^*A}y_1 \cup \overline{A^*A}y_2 \cup \cdots \cup \overline{A^*A}y_m. \end{aligned}$$

To prove the contrapositive, we show, by induction on expression length, that the language of any finite-order expression can be expressed as a finite order language, $A^*F \cup G$, where F and G are finite. The basis is trivial: $A^* = A^*\{\lambda\} \cup \{\}$. For the inductive step, consider a finite-order expression that is composed of two shorter finite-order expressions α and β, which by induction hypothesis have corresponding finite sets $F_\alpha = \{x_1, \ldots, x_n\}$, $F_\beta = \{y_1, \ldots, y_m\}$, $G_\alpha = \{v_1, \ldots, v_r\}$, and $G_\beta = \{w_1, \ldots, w_s\}$. There are four cases:

$$\begin{aligned} L(\alpha \cdot a) &= A^*(F_\alpha \cdot \{a\}) \ \cup \ G_\alpha \cdot \{a\}. \\ L(\alpha \cup \beta) &= A^*(F_\alpha \cup F_\beta) \ \cup \ (G_\alpha \cup G_\beta). \\ L(\alpha \cap \beta) &= A^*F \cup G, \\ &\qquad \text{where } G = \bigcup_{\substack{1 \le i \le n \\ 1 \le k \le r}} (A^*x_i \cap v_k) \cup \\ &\qquad\qquad \bigcup_{\substack{1 \le j \le m \\ 1 \le l \le s}} (A^*y_j \cap w_l) \cup \\ &\qquad\qquad \bigcup_{\substack{1 \le k \le r \\ 1 \le l \le s}} (v_k \cap w_l), \\ &\qquad\qquad \text{and } F = \{x_i \in F_\alpha : \exists y_j \in F_\beta \text{ s.t. } y_j \text{ is a suffix of } x_i\} \cup \\ &\qquad\qquad\qquad \{y_j \in F_\beta : \exists x_i \in F_\alpha \text{ s.t. } x_i \text{ is a suffix of } y_j\}. \\ &\qquad \text{The definition of } F \text{ follows from the fact that} \\ &\qquad\qquad A^*F = \bigcup_{\substack{1 \le i \le n \\ 1 \le j \le m}} (A^*x_i \cap A^*y_j). \\ L(\overline{\alpha}) &= \overline{A^*x_1 \cup A^*x_2 \cup \cdots \cup A^*x_n \cup v_1 \cup v_2 \cup \cdots \cup v_p} \\ &= \overline{A^*x_1} \cap \overline{A^*x_w} \cap \cdots \cap \overline{A^*x_n} \cap \overline{v_1} \cap \overline{v_2} \cap \cdots \cap \overline{v_p}, \end{aligned}$$

$$
\begin{aligned}
\text{where } \overline{v_i} &= A^*Av_i \cup \overline{A^*v_i}, \\
\overline{A^*x} &= \{\}, \text{ if } x = \lambda; \\
\overline{A^*x} &= \overline{A^*a_1a_2\cdots a_z}, \text{ if } x = a_1a_2\cdots a_z \\
&= \{\lambda\} \cup A^*\overline{a_z} \cup A^*\overline{a_{z-1}}a_z \cup \cdots \cup A^*\overline{a_1}a_2\cdots a_z, \\
&\quad \overline{a} = \{b \in A : b \neq a\}.
\end{aligned}
$$

The final expression for $L(\overline{\alpha})$ shows that the finite order expression $\overline{\alpha}$ can be transformed into a finite-order expression constructed from single symbols and A^* using only concatenation, intersection, and union; that is, an expression for which the induction is already proved.□

The fact that DMC models are finite-order may now be expressed clearly:

Corollary 4.5.1 *The language of a state s_i in a DMC model M_k, $C_k(s_i)$, is finite-order for all i and k.*

Proof: The recursive definition of $C_k(s_i)$ reduces to a finite-order expression, that is,

$$C_k(s_i) = \mathbf{L}(s_i) \cap \bigcap_{s_d \in \mathbf{extns}_k(s_i)} \overline{\mathbf{L}(s_d)}.$$

4.5.3 DMC Models Are Not FSMX

It is instructive to consider restricted DMC embodiments in which $\mathcal{E}()$ allows transitions to be redirected only once and disallows the redirection of transitions from states that already have extensions. In such models, Equation (4.10) holds for every state. In that case,

$$C_k(s_i) = A^*\mathbf{context}(s_i) - \bigcup_{s_d \in \mathbf{extns}_k(s_i)} A^*\mathbf{context}(s_d),$$

which is logically equivalent to the closed-form transition function of FSMX models.

FSMX models, defined by Rissanen [Ris83], are characterized by their state-set and transition function:

- The states satisfy a *suffix property*, where states mapped to unique finite strings, or conditioning contexts, and for every state with context x, there exist states for each proper suffix of x. The set of maximal conditioning contexts that have states in the model is determined by a growth heuristic, and varies among different techniques.

- The transition function has a closed form based on state conditioning contexts. The next state is always the state whose associated finite context is the maximal suffix, relative to the context partition defined by the model, of the already-processed portion of the input sequence.

DMC transition functions cannot, in general, be expressed in such a simple closed form.

When $\mathcal{E}()$ allows transitions to be redirected more than once, distinct states s_i and s_j may be mapped by the function **context**() to the same string:

$$\begin{aligned} \mathbf{context}(s_i) = \mathbf{context}(s_j) \quad \Rightarrow \quad & \exists\ l < |\mathbf{context}(s_i)| \text{ s.t.} \\ & \mathbf{prefix}^l(s_i) = \mathbf{prefix}^l(s_j), \end{aligned}$$

where $\mathbf{prefix}^l()$ is the function obtained by composing the function **prefix**() with itself $l-1$ times.

Furthermore, a completely independent complication arises when transitions from a state that already has extensions may be redirected. In this case, the state corresponding to $M_k(wa)$ will not always be the state in M_k whose **context**() is the longest match to the end of wa. An example of this situation occurs with respect to an arbitrary state s_i in an arbitrary model M_k, where:

$$\begin{aligned} \mathbf{context}(s_i) &= \text{`abc'}, \\ \mathbf{context}(\mathbf{prefix}(s_i)) &= \text{`ab'}, \\ \mathbf{context}(s_d) &= \text{`wabc' for some } s_d \in \mathbf{extns}_k(s_i), \\ \mathbf{context}(\mathbf{prefix}(s_d)) &= \text{`wab', and} \\ \mathbf{context}(s_r) &= \text{`vwab' for some } s_r \in \mathbf{extns}_{d-1}(\mathbf{prefix}(s_d)). \end{aligned}$$

Figures 4.3 and 4.4 are included to illustrate this type of situation in two different ways. Figure 4.3 illustrates the general structure of any DMC model, and shows why the alternating recursion of the characterization is necessary. It can also be used here as a template for visualizing concrete examples, the example above in particular, by going through the exercise of actually writing the given contexts on the states with the above labels. Figure 4.4 shows how such a situation can arise (using a necessarily simpler example) by growing a small DMC model that cannot be emulated by an FSMX model, from scratch. In Figure 4.4, the transition from the state with context 'aab' goes to a state with context 'bc,' although a state with context 'abc' is available. In an FSMX model, the 'abc' state would have been entered instead.

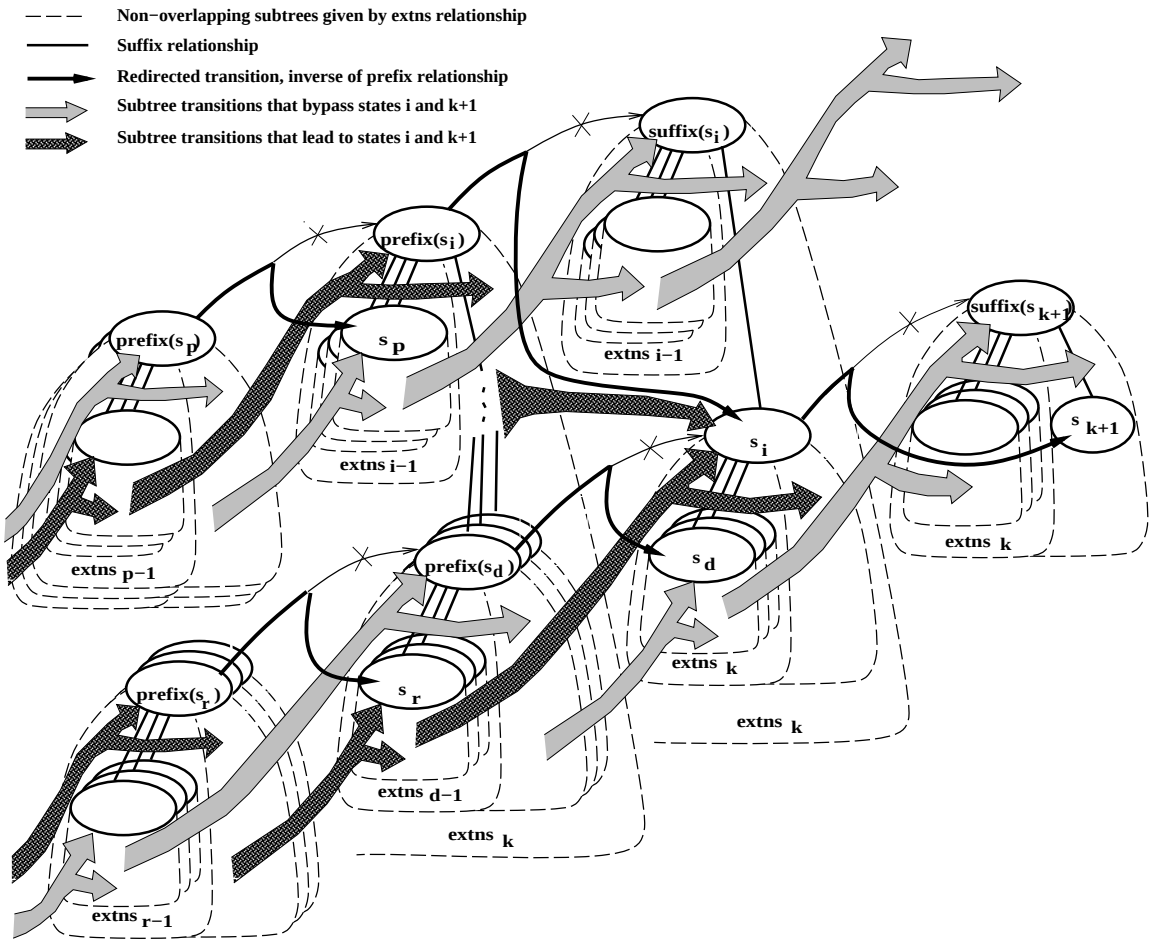

Figure 4.3: **Example-independent illustration** of the (necessary) alternating recursion of function $\mathbf{L}()$. The states in the upper subtrees correspond to contexts in the first term of $C_k()$, which describes strings that bring M_k to the subtree headed by s_i. The states in the lower subtrees correspond to contexts in the second term of $C_k()$, which describes strings that have been directed away from s_i into s_i's subtree. Note, the figure does not illustrate multiply-redirected transitions.

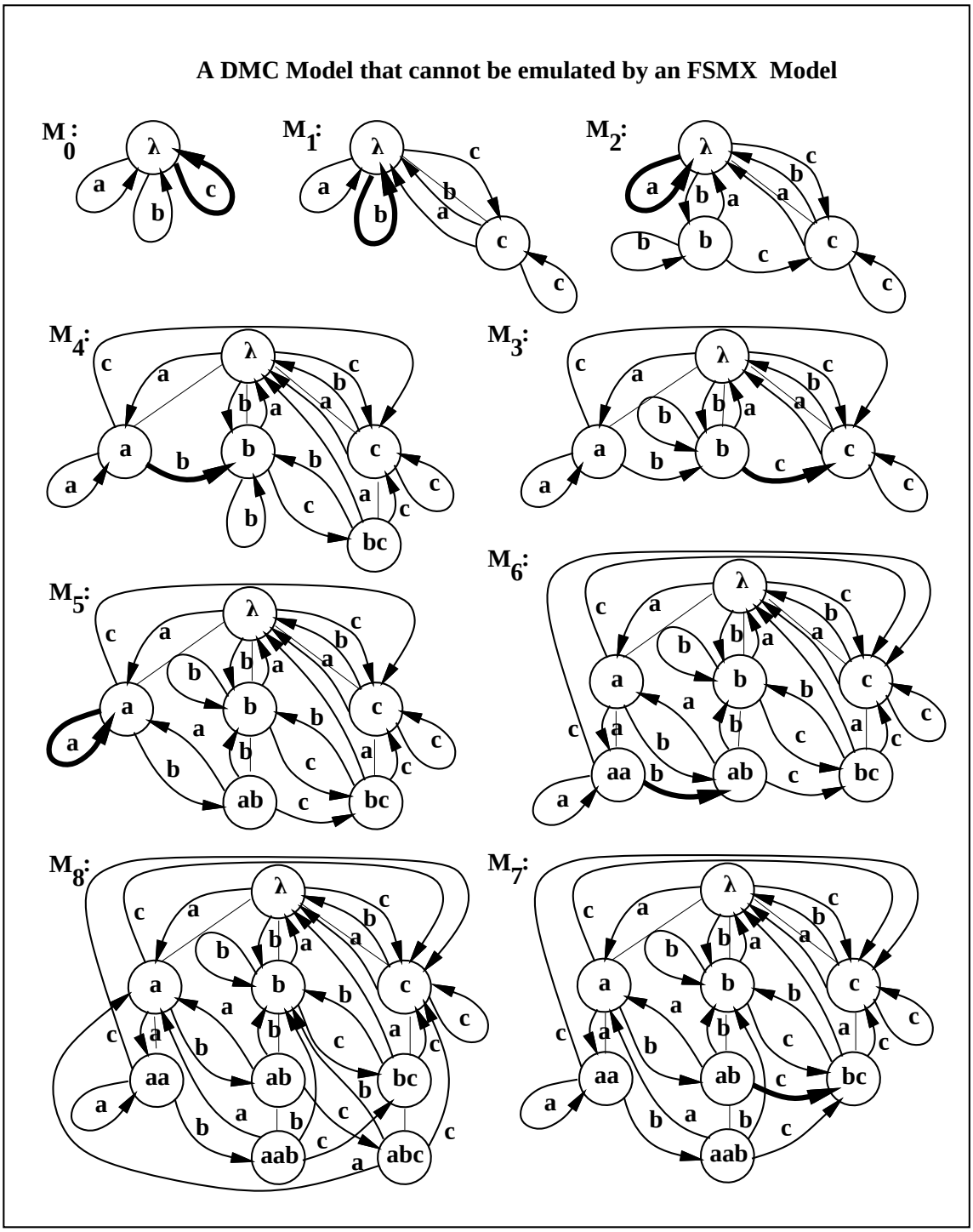

Figure 4.4: **DMC models cannot be emulated by FSMX models**. The temporal order in which contexts corresponding to states in the model are extended by adding children affects the context partition in DMC models. In contrast, only the occurrences of the contexts themselves, and not the order in which they occur, affects FSMX structure. For example, on input 'c,' model M_8 will go from the state with context 'aab' to the state with context 'bc', although the state labeled 'abc' is present. An FSMX model would enter the state labeled 'abc'.

Here, **context**(s_d) is the best-matching context for any input substring in ending in 'vwabc'; however, substrings ending in 'vwabc' take M_k to s_i, not s_d. This example proves that DMC states cannot be uniquely characterized with single strings. Thus a stronger characterization is required for DMC models than for FSMX models, because DMC models can recognize languages for which no FSMX model exists. Furthermore, for any FSMX model there exists a linguistically equivalent DMC model. Another interesting consequence of the fact that DMC is not FSMX is that this proves conclusively that the family of languages generated by FSMX models does not properly contain the languages generated by (finite order) Markov Models. This contradicts the conventional wisdom that FSMX models "generalize the class of Markov Models" [Ris86a].

4.5.4 Structural Comparison

The **context**() of a DMC state is analogous to the finite contexts of the states in FSMX models [CW84b, CTW95, Fur91, RC89, Ris83, Ris86a, WLZ92, Wil91] vis à vis the terms of the definition of $C_k()$, and the fact that every string that takes a DMC model to a given state s_i ends in **context**(s_i). Both DMC and FSMX models are tree-structured, where children states correspond to minimal extensions of their parent's context. Thus DMC and FSMX models are similar enough to allow meaningful comparison of their abstract structural differences, in terms of conditioning contexts.

Since FSMX models are characterized by a uniformly restricted version of DMC's characterizing function $C_k()$, we know that any FSMX model can be simulated by a DMC model. However, such a simulation will require extra states in the DMC model, for the class of DMC models does not contain the class of FSMX models (or *vice versa*). DMC models are strictly Markovian, that is, the next state is always a function of the current state and the currently scanned symbol; whereas the next state in an FSMX model may also depend upon which states were visited before the current state. Note that it is equivalent to say that DMC models satisfy a *prefix* property: for every state s there exist states p for each proper prefix of **context**(s) such that $\mu_k(p, w) = s$, where **context**(s) = **context**(p) $\cdot\, w$. A DMC model that simulates an FSMX model, a Markovian FSMX model, would satisfy both the prefix property above, and the suffix property satisfied by FSMX models. The comparison between such a model and an arbitrary DMC model explores structural differences

besides the Markov (or prefix) property.

The first difference is that a DMC model can have distinct states whose associated conditioning context partition elements have the same maximum common suffix. The function **context**() describes these maximum common suffixes. FSMX models also associate conditioning context partition elements with each state, but each is fully characterized by its maximum common suffix, and therefore there is no duplication of the maximum common suffix of any state. The duplication in DMC is partially caused by redirecting the original transition for which a given state was created (that state's *prefix transition*). Any benefit of redirecting a prefix transition is realized only if the other transitions entering the given state, or earlier crossings of the prefix transition itself, have caused the frequency distribution on the state's outgoing transitions (described in Section 6.5) to become non-representative. The cost of redirecting prefix transitions in the absence of such contamination is pervasive model redundancy. Not only can this hinder structural convergence, but the convergence of statistical parameters may be slowed because message statistics are distributed unnecessarily.

A more important distinction is DMC's capacity for variable-length minimal extensions of a context. This capacity alone sets DMC's (Markovian) data model apart from all FSMX models. Techniques that build FSMX models [CW84b, CTW95, Fur91, RC89, Ris83, Ris86a, WLZ92, Wil91] all grow models with *single-character* minimal extensions. Thus DMC has the capacity for modeling the most probable substrings in a given sequence using fewer states than any of its counterparts.[4]

The remaining structural difference is due to the fact that the (temporal) order in which finite substrings in a given sequence become recognizably frequent strongly affects the structure of the regular sets of conditioning contexts associated with each state. Thus the structure of DMC models reflects higher-order statistics of the conditioning contexts themselves. In contrast, the structure of FSMX models only reflects the zero-order statistics of the conditioning contexts (i.e., how many times each context appears in the source message). Therefore, DMC's Markov models record information about the input sequence that FSMX models do not.

The following are immediate corollaries of our main result:

---

[4] That is, assuming the sequence was generated by a Markov FSMX source. Using a Markov model such as DMC's to simulate any non-Markovian FSMX source requires systematic addition of extra states.

1. DMC states have a defined, locally-variable minimum order.
2. Every state has the same or higher minimum order than its parent.
3. The act of invoking the criterion $\mathcal{E}()$ to decide when to add a state s_{k+1} as an extension to a given s_t is indeed local order estimation.

4.6 Curbing Counterproductive Model Growth

An obvious way to reduce the number of states in DMC models is to use a larger input alphabet. However, as originally presented, DMC is only feasible with a binary alphabet, since $|A|$ outgoing transitions are created for every new state. Other authors [TR93, Ton93, Whi94, Yu93] have independently generalized DMC to larger alphabets using variations of what we call *lazy cloning*, which copies outgoing transitions only as needed. However, only the lazy-evaluation solutions introduced in [Bun94] and in [TR93] successfully reduce DMC's memory requirements without eliminating DMC's advantages. (Sections 6.5 and 9.5 explain and refine the ideas from [Bun94] and Section 9.5 compares their performance with the technique of [TR93].) The others all had a similar approach: when the required outgoing transition was absent from the current state, the needed transition was copied from a state that was essentially a zeroth or first-order state.

Our analysis implies that the best state from which to copy outgoing transitions one at a time is the same state they would have been copied from if the copying were done all at once: the **suffix**() state. This way, the transitions go to a next state with the longest possible matching conditioning context. Copying outgoing transitions from a low-order state has the opposite effect. Thus our analysis of DMC explains the successes and failures of the various practical approaches.

Several natural solutions for curbing DMC's counterproductive model growth follow from our characterization of DMC. They apply individually and in combination:

1. $\mathcal{E}()$ should better approximate entropy-based local order estimation. The frequency of a particular edge does say something about the probability of the state it leads to. Thus the original $\mathcal{E}()$ crudely approximates the contribution of that destination state to the entropy of the model. The other approximation extreme is exemplified by Rissanen [Ris86a] and Furlan [Fur91]. These authors closely approximate the relative entropies between states and their minimal

extension states using counters which keep track of the code-length differences.

2. The refinement eligibility criterion $\mathcal{E}()$ should prevent transitions from being redirected more than once, under most circumstances.[5]

3. The transitions exiting any given state should be copied from its **suffix**() one at a time, as they are needed. When a state does not have the required outgoing transition, the probability estimate can be made from its **suffix**(). This way, larger input alphabets can be economically accommodated, since only conditioning contexts that have been seen before will be represented by the model. Probability estimation can proceed using the recursive blending technique of PPM [CW84b], or by lazy evaluation of DMC's initial frequency distribution assumption. For example, if blending method PPMC [Mof90] were used, the frequency on the **suffix**() edge should always equal the number of distinct symbols which have been seen when in a given state, and the application of such optimizations as *exclusion* and *update exclusion* [BCW90] is straightforward. DMC's probability estimation technique, and some efficient variations, are described in Chapter 6.

4.7 Summary

This chapter's analytical study of DMC completes our study of model structure. Chapter 3 showed how to construct suffix tree models using only the "splitting" of *string*-labeled transitions. This chapter showed how to construct suffix tree models using only the redirection of *symbol*-labeled transitions, and proved that the classes of models built by these two operations are distinct, albeit overlapping. The remaining technical chapters of this thesis cover the computations required for estimating probabilities using any suffix-tree-structured model.

---

[5] The tradeoff between prediction quality and space efficiency depends upon the nature of the refinement criterion $\mathcal{E}()$. J. Teuhola and T. Raita report [personal correspondence] that prediction quality degrades when this restriction is combined with lazy cloning. In [TR93] they based $\mathcal{E}()$ only upon the difference between a candidate transition's frequency and the frequency of the transition's destination state.

Chapter 5

FREQUENCY UPDATES

How should we organize frequency data at the suffix-tree states? An obvious way to have the value of **count**$[a, s]$ reflect the characteristics of the input sequence is to have it equal the number of times the symbol a has occurred when state s was excited in the past. We shall refer to this technique as *full updates*. A different approach, called *update exclusion* [Mof90], so effectively and consistently improves the probability estimates produced by PPM that its application and combination with state selection in all suffix-tree models, including PPM*, warrants careful consideration and, as we explain, some modification for combination with state selection.

5.1 Model Semantics II: Update Exclusion vs. Full Updates

Generally speaking, full updates correctly update the frequencies at every FSM model that is simulated by the suffix-tree, while update exclusion correctly updates the frequencies at the single FSM model that corresponds to the maximum suffix-tree frontier. As defined in [Mof90], *update exclusion* is a frequency-update technique that increments the counts of event $a|s$ when s is excited and either symbol a is novel at state s, s is the parent of a state where a is novel, or s is the maximum-order excited state. Thus all ancestors of the maximum-order excited state that recognizes the current symbol are excluded from the update process. In addition to improving compression performance, update exclusion can reduce execution time.

Update exclusion improves the performance of blending in context models, but it renders the models incompatible with state selection by changing their semantics. In all suffix-tree models, update exclusion affects the context partition element that conditions the estimated probability, $P_e(a|s_j)$, of an event a given the frequency data at state s_j. Without update exclusion, $P_e(a|s_j)$ is conditioned by the strings in $L(s_j)$, which equals the conditioning context partition element of the entire subtree rooted

by state s_j. That is,

$$L(s_j) = \begin{cases} A^*\mathbf{context}(s_j) & \text{if the model is FSMX,} \\ C_{j-1}(\mathbf{prefix}(s_j) \cdot \mathbf{symbol}(s_j)) & \text{if the model is created by transition-redirection, as are DMC models,} \end{cases}$$

where $C_{j-1}()$, **prefix**(), **symbol**(), and **context**() are defined as in Chapter 4. On the other hand, with update exclusion, the estimate is conditioned by a conditioning context partition element that approximates

$$L(s_j) - \bigcup_{\{k:\mathbf{suffix}(s_k)=s_j\}} L(s_k).$$

(It would equal the above set of strings if not for the fact that the value of **count**$[a, s_j, 1]$ is updated every time a child of state s_j sees symbol a for the first time.)

Although update exclusion improves PPM's performance, the original PPM* implementation did not employ update exclusion. Update exclusion would interfere with PPM*'s original state-selection mechanism if the updates were implemented in PPM* as they are in PPM. This is because PPM* does not always compute the coding distribution from the frequency data at the maximum-order excited state. Instead, PPM* *selects* the lowest order virtual (or deterministic) excited state when any virtual states are excited. This is a good policy because it always selects the excited virtual state that will produce the lowest-entropy probability distribution: all simultaneously excited virtual states v have seen the same single event, $a|v$, at least once; the lowest order excited virtual state will have seen a at least as many times as any of its descendants; and the excited virtual states are always the highest order excited states.

As we explain in Section 7.4, selecting a state is semantically equivalent to selecting an entire model, which requires pretending that all descendants of the selected state never existed. Without update exclusions, the value of **count**$[a, s]$ is the same as if s had no descendants in the suffix tree. However, with update exclusions, the counts for all symbols recognized at the selected state s will be too small, if s has children.

5.2 Update-Exclusion Techniques for use with State Selection

To apply update exclusion to suffix-tree FSMs (e.g., PPM*) in a way that is compatible with state-selection (including PPM*'s original state-selection mechanism), we emulate the effect that update exclusion has on the coding distributions computed, dynamically, on a *per-node* basis, instead of globally.

5.2.1 Dual Frequency Updates

Let X be a global binary parameter such that $X = 1$ if the update exclusion option is enabled for the model, otherwise, $X = 0$. Let s' be a variable that denotes which, if any, of the currently excited states has been specifically selected as the source of the *coding distribution*, which will be computed from some combination of the excited states of Markov order equal to or less than the order of s'; when no excited state has been specifically selected, s' equals *null*. We will use $u : S \rightarrow \{0, 1\}$ to denote whether we use update-excluded frequencies at a given state s, where

$$u(s) = \begin{cases} 0 & \text{if } s = s' \\ X & \text{otherwise.} \end{cases}$$

We shall keep two frequency counts for each event: PPM*'s update-excluded frequency counter, $\mathbf{count}[a, s, 1]$, always equals the number of times that a is novel at state s or any child of s, and $\mathbf{count}[a, s, 0]$ always equals the number of times a has been seen when any state of the subtree rooted by s has been excited. Note that in internal nodes of PPM, or any node of PPM*, $\mathbf{count}[a, s, 1] \leq |A| + 1$, where A is the input alphabet. So, PPM*'s unbounded order constrains the value of $\mathbf{count}[a, s, u(s)]$ to be a function of the suffix-tree structure itself. In PPM* models, $\mathbf{count}[a, s, 1]$ equals the number of children of s with out-events labeled a, plus 1, if a was seen at state s before it was seen at s's children. Similarly, $\mathbf{count}[a, s, 0]$ equals the number of (descendants) of s with out-events labeled a, plus 1, if a was seen at state s before it was seen at s's descendants. Furthermore, since the counter values are a function of the model structure, explicit counters are not strictly necessary in a PPM* implementation.

In general however, on-line suffix-tree models that combine blending or mixtures with state-selection will require the dual frequency counts $\mathbf{count}[a, s, 1]$ and $\mathbf{count}[a, s, 0]$. Henceforth, we shall use the dual notation.

5.2.2 Maximum-Order Updates

The combination of state-selection and update exclusion opens the possibility for a third update technique for use with state selection, *max-order updates*. When max-order updates are enabled, $\mathbf{count}[a, s, 1]$ is incremented only if s is the maximum order excited state. Max-order updates are interesting because they will cause the conditioning context partition element to actually *equal* the context class that is approximated by update exclusion:

$$L(s_j) - \bigcup_{\{k:\mathbf{suffix}(s_k)=s_j\}} L(s_k).$$

We hypothesized that max-order updates are the best update technique for computing mixtures with state selection, just as update exclusions have been the best for computing mixtures without state selection. State selection dynamically prunes the suffix tree at the selected state s', with the effect that all frequencies in the entire subtree rooted by s' are viewed as belonging to s'. Thus, with state selection, max-order frequency updates are completely visible to ancestor states. However, without state-selection, max-order frequency updates would not be visible to all ancestors, and regular update exclusions compensate for this. The use of max-order updates causes states to have zero frequencies for events that have occurred before, but whose frequency increment was given to a proper descendant.

Preliminary experiments with max-order updates supported our conclusion that update exclusion approximates the max-order update context partition, however, there was no performance advantage. Max-order updates are not considered further in this thesis, except as an option required in our taxonomy to exactly emulate the DMC variant GDMC [TR93].

5.3 Summary

The principal update schemes given here, *full updates*, *update exclusion*, and *max-order updates*, form a single set of interchangeable options for the executable taxonomy and are empirically evaluated in Chapter 9. Update exclusion has been very successful in improving the coding distribution computations of PPM variants, which historically do not employ information-theoretic state selection. Regardless of which of the update options is selected for organizing the frequency data that are used to

estimate the coding distribution (the topic of next chapter), only frequency distributions obtained using full updates are appropriate for performing information-theoretic state selection (Chapter 7). Thus, the best frequency update schemes for computing the coding distribution are distinct from the ideal update scheme for state selection. This problem is resolved by using dual frequency counts for each event at each state, whenever state selection is combined with a coding distribution computation that requires an update scheme other than full updates.

Chapter 6

ESTIMATING THE CODING DISTRIBUTION

The preceding chapters have presented frequency update options for any suffix tree, and have shown the structural similarities between PPM, PPM*, and DMC models by transforming them each to a suffix-tree representation with an explicit mapping between states and conditioning contexts. The next two chapters address the following questions concerning the excited states of any suffix-tree model:

- Which excited state's frequency data will provide the best probability estimate?
- How can we combine the frequency data at the excited states to provide the best probability estimate?

This chapter answers the second question, with a general technique that we call "mixtures," which we decompose into two components: a *mixture weighting function* and an *inheritance evaluation time*.

The best-performing method in the data compression literature for computing probability estimates of sequences on-line using a suffix-tree model is the *blending* technique used by PPM [CW84b, Mof90]. Blending can be viewed as a bottom-up recursive procedure for computing a *mixture*, barring one missing term for each level of the recursion, where a mixture is basically a weighted average of several probability estimates. We shall show by decomposition that mixtures generalize the techniques used in DMC variants [CH87, TR93], as well as PPM variants, and thus these techniques, along with other variants of mixtures, are interchangeable.

6.1 Recursive Mixtures

We are concerned with estimating a probability $P_e(a_i|a_1a_2\cdots a_{i-1})$ using the frequencies stored at the *excited* states of a suffix-tree FSM (see Chapter 2 of [Bun96]), where the excited states are those states of the FSM whose associated conditioning context partitions contain the sequence $a_1a_2\cdots a_{i-1} \in A^*$. At any time, the excited states of a suffix-tree FSM are linked, at least abstractly, by an unbroken chain of suffix

pointers, which, for a given state s, point to the state with the smallest conditioning context that properly contains the conditioning context of s.

Let s_0 and s_{-1} be the order 0 and order -1 states of a suffix tree, respectively, and define $\alpha(s)$ to be the number of times a novel event has occurred at a given state s. That is,

$$\alpha(s) = |\{a : \mathbf{count}[a, s, u(s)] > 0\}|,$$

where $a \in A$, the finite input alphabet, and $u : S \rightarrow \{0, 1\}$ selects between full-update frequencies given by **count**[-,s,0] and update-excluded frequencies given by **count**[-,s,1] at state s. Thus,

$$u(s) = \begin{cases} 0 & \text{if } s = s', \text{ the selected state;} \\ X & \text{otherwise,} \end{cases}$$

where X is a global binary variable denoting whether *update exclusions* are enabled for the model. Note that $\mathbf{count}[a, s, 1] = 0 \Leftrightarrow \mathbf{count}[a, s, 0] = 0$, so the value of $u(s)$ does not affect the value of $\alpha(s)$. Let $\mathbf{count}(a, s) = \mathbf{count}[a, s, u(s)] + k$, where k is the initial frequency value (ideally, $k = 0$), and k is a global constant that remains fixed for the lifetime of the model. Lastly, let the node-count function $\mathbf{count} : S \rightarrow R$ be defined as follows:

$$\mathbf{count}(s) = \sum_{a:\mathbf{count}[a,s,u(s)]>0} \mathbf{count}(a, s).$$

Given the above definitions, a simple bottom-up procedure for recursively computing a mixture that estimates the probability of a given event, $a_i = a$, starting from an excited state s, is

$$P_e(a|s, i) = \begin{cases} W(s) \cdot \frac{\mathbf{count}(a,s)}{\mathbf{count}(s)} + (1 - W(s)) \cdot P_e(a|\mathbf{suffix}(s), h(s, a, i)) & \text{if } s \neq s_{-1} \\ \frac{1}{|A|+1-\alpha(s_0)} & \text{otherwise} \end{cases}$$

where $0 \leq W(s) < 1$, and $h(s, a, i) \leq i$.

Assuming the mixture computation is initiated at the maximum-order excited state, this procedure computes a mixture of maximum-likelihood probability estimators for all excited states, except for the order -1 state, which contributes an explicitly assumed initial distribution that must be non-zero for all possible symbols in A.

The *inheritance evaluation time* $h(s, a, i)$ defines when, relative to the input sequence, the ancestor's contribution to the mixture, $P_e(a|\mathbf{suffix}(s), h(s, a, i))$, is computed. The recursive *mixture weighting function* $W()$ determines the degree of influence the ancestor's contribution will have relative to the contribution of the frequencies local to state s.

There are two essential questions that must be addressed when defining an effective mixture:

- How do we define the *mixture weighting function* $W()$?
- How do we define the *inheritance evaluation time* $h(s, a, i)$?

6.2 Mixture Weights

Our goal is to define an easily computable weighting function $W : S \rightarrow [0..1)$ that will cause $P_e(a|s)$ to assign the greatest likelihood to the currently scanned symbol, on average. This implies, for one thing, that our weighting function should assign a low value to $W(s)$ whenever it is likely that the currently scanned symbol corresponds to an event that has never occurred when s was excited, so that the weight $1 - W(s)$ of the ancestors' contribution to the mixture is relatively high. Thus the choice of weighting function reduces to a solution to an ancient problem—the "zero-frequency problem," or how to assign a likelihood to an event that has never occurred before—for which it is widely agreed that no principled solution exists, in the absence of *a priori* knowledge [WB91]. Therefore, the merit of any weighting function for a universal model is determined analytically by how the assumptions it imposes interact with other assumptions made in the model, and empirically by its performance on actual data.

Several approaches to solving the zero frequency problem, known as "escape" mechanisms, have been used successfully with PPM implementations. Four of the simplest and best-performing escape mechanisms are known in the literature as 'A,' 'B,' 'C,' and 'D' [WB91]. In this section, we shall show how these simple escape mechanisms correspond to different weighting functions $W(s)$. We introduce a general formula for $W(s)$ that relies upon global changes to the initial values of event frequencies to express each of these escape mechanisms exactly, and which allows efficient implementation of the mixture computation.

6.2.1 Mixture Weights with Variable Initial Frequencies

Each of the escape mechanisms 'A'–'D' can be described exactly as a general weighting function $W(s)$, where

$$W(s) = \frac{\mathbf{count}(s)}{\mathbf{count}(s) + \frac{\alpha(s)}{d(s)}},$$

if we let the escape mechanism determine the global constant k (which is ideally zero) such that $\mathbf{count}(a, s) = \mathbf{count}[a, s, u(s)] + k$ if $\mathbf{count}[a, s, u(s)] > 0$, and $\mathbf{count}(a, s) = 0$ if $\mathbf{count}[a, s, u(s)] = 0$. This way, the four escape mechanisms are given by the following assignments to $d(s)$ and initial frequency value k:

A: $d(s) = \alpha(s)$ $\quad k = 0$
B: $d(s) = 1$, $\quad k = -1$
C: $d(s) = 1$, $\quad k = 0$
D: $d(s) = 2$, $\quad k = -\frac{1}{2}$.

What assumptions about the input data do these choices of weighting formulas impose? Each of these weighting functions base the weights on the number of times in the past that a node s has "missed," that is, failed to assign a non-zero likelihood to the scanned symbol when excited. The key difference among the escape mechanisms is how much emphasis is placed on the predictions conditioned by excited low order states relative to the predictions conditioned by excited high order states.

In the general weighting formula for $W(s)$ above, more emphasis is placed on higher order states as $d(s)$ increases in numerical value. Thus, if we sort the escape mechanisms by increasing values of $d(s)$, we get 'B' $\preceq$ 'C' $\preceq$ 'A' $\prec$ 'D', when $\alpha(s) < 2$ and 'B' $\preceq$ 'C' $\prec$ 'D' $\preceq$ 'A', otherwise. With larger values of k, more emphasis is placed upon higher order states. Thus if we sort the escape mechanisms by increasing values of k, we get 'B' $\prec$ 'D' $\prec$ 'C' $\preceq$ 'A'. Clearly 'B' places the lowest relative emphasis on high order states, while 'A' tends to place the greatest emphasis on high order states. Mechanisms 'D' and 'C', which consistently and significantly outperform 'A' and 'B' in practice, are somewhere in the middle. Method 'D' systematically favors deterministic states (i.e., states that recognize only one input symbol—they are always among the highest order excited states), and tends to slightly outperform 'C'.

One clear benefit of these particular escape mechanisms is that they simplify the

algebra required to compute the mixture. With escape formulas 'A'–'D', the general mixture formula becomes:

$$P_e(a|s,i) = \begin{cases} \frac{\mathbf{count}(a,s)+\left(\frac{\alpha(s)}{d(s)}\cdot P_e(a|\mathbf{suffix}(s),h(s,a,i))\right)}{\mathbf{count}(s)+\frac{\alpha(s)}{d(s)}} & \text{if } s \neq s_{-1} \\ \frac{1}{|A|+1-\alpha(s_0)} & \text{otherwise} \end{cases}$$

where $0 \leq W(s) < 1$.

6.2.2 Inherited Frequencies

In general, given any of the weighting formulae above, we can express the mixture as

$$P_e(a|s,i) = \begin{cases} \frac{\frac{\text{numerator}(W(s))}{\mathbf{count}(s)}\cdot\mathbf{count}(a,s)+I(a,s,i)}{\text{denominator}(W(s))} & \text{if } s \neq s_{-1} \\ \frac{1}{|A|+1-\alpha(s_0)} & \text{otherwise,} \end{cases}$$

where $I(a,s,i) = (\alpha(s)/d(s)) \cdot P_e(a|\mathbf{suffix}(s), h(s,a,i))$. $I(a,s,i)$ is the *inherited frequency* for event $a|s$ at time i. The next section covers the computation of $I(a,s,i)$ and how different computation times affect the model.

6.3 Inheritance Evaluation Times

There is a spectrum of evaluation times for inherited frequencies, which denote when $\alpha(s)/d(s) \cdot P_e(a|\mathbf{suffix}(s), h(s,a,i))$ is computed, relative to the lifetimes of state s and event $a|s$. We use *inheritance evaluation time* $h(s,a,i)$ to explicitly specify $P_e(a|\mathbf{suffix}(s))$ as a function of the frequency data that are available at one of the following times:

- *inherit at model creation:* $P_e(a|\mathbf{suffix}(s), h(s,a,i))$ is computed when the initial model consisting of state s_{-1} is created; $h(s,a,i) = 0$.
- *inherit at state creation:* $P_e(a|\mathbf{suffix}(s), h(s,a,i))$ is computed when state s is added to the model; $h(s,a,i)$ equals the length of the input sequence that had been processed so far when s was added to the model.
- *inherit before novel event update:* $P_e(a|\mathbf{suffix}(s), h(s,a,i))$ is computed when novel event $a|s$ first occurs, before its frequency is incremented; $h(s,a,i)$ equals the length of the input sequence that had been processed so far when a occurred for the first time when s was excited.

- *inherit at every event visit:* $P_e(a|\mathbf{suffix}(s))$ is (re)computed each time event $a|s$ occurs; $h(s,a,i) = i$.

Inheritance evaluation time is a global option that remains fixed for the lifetime of the model. In the remainder of this work we will usually simplify the notation by making evaluation times i and $h(s,a,i)$ implicit, thus replacing $P_e(a|s,i)$, $I(a,s,i)$, and $P_e(a|\mathbf{suffix}(s), h(s,a,i))$ with $P_e(a|s)$, $I(a,s)$, and $P_e(a|\mathbf{suffix}(s))$ respectively.

There are two important features that must be considered in the choice of an inheritance evaluation time. The computational and memory cost of its implementation, and what the inheritance evaluation time assumes about the data, relative to the other times. Once the relative differences in inheritance evaluation times are understood, the model designer can intelligently trade off the appropriateness of the assumption about the target data versus the space and time requirements of an implementation.

6.3.1 Inheritance Evaluation Times in Practice

Probably the most natural time for computing inherited frequencies $I(a,s)$ is every single time the state s becomes excited: in this case, the mixture is simply a weighted average of the probabilities estimated from the current frequency data at each excited state. In [BCW90] this approach is called "full blending." However, computing such weighted averages is expensive, and computations cannot be reused between visits to a given set of excited states. Furthermore, there was no published evidence prior to this work that it produces better probability estimates—no published on-line algorithms use it. The remaining alternatives generally allow faster implementations.

At the other extreme, *inherit at model creation* corresponds to adding a constant assumed initial frequency distribution to the observed frequencies at any given node. This approach leads to simple analyses and does not affect asymptotic convergence. Thus it is employed by most theoretical constructions that use information-theoretic state-selection. However, our experiments show that the probability estimates produced by using this inheritance evaluation time are not competitive with the other inheritance times considered here, even when they are combined with state selection.

The DMC algorithm, which originally used a binary alphabet, adds each new state to its model by "cloning" an eligible parent state. Each clone receives a scaled copy of the parent state's frequency distribution the moment it is added. Since, as we proved

in Chapter 4 of [Bun96], the conditioning context relationship among clones and parent states is equivalent to that among suffixes in other suffix-tree models, *inherit at state creation* corresponds to the numerical aspects of cloning. For non-binary input alphabets and aggressive model growth heuristics, evaluating every symbol's inherited frequency at every new state is prohibitive, in terms of memory and computation costs, as was historically demonstrated with larger-alphabet parameterizations of the DMC algorithm. We include this inheritance time only for completeness, and do not evaluate its performance.

Overall, the best approach practically, and performance-wise, is the evaluation of inherited frequencies whenever a novel event a is seen at state s. This is similar to blending in PPM variants, and can also be used in lazy implementations of large-alphabet DMC variants.

6.3.2 The Significance of Inheritance Evaluation Time

Basically, inheritance evaluation times select the degree to which *recent vs. relatively historical* event frequencies that are conditioned by a given set of contexts predict the behavior of the events conditioned by proper subclasses of those contexts. In contrast, mixture weighting functions determine the degree to which *inherited vs. observed* event frequencies predict the behavior of events conditioned by a given set of contexts.

There are more direct and quantifiable ways of establishing how much more recent events should matter than events that happened long ago. For example, a sliding window of input history can be kept, and as sequence symbols that have passed through the buffer pass out of the buffer, the event frequencies originally incremented by these symbols can be decremented [Wil91]. Alternatively, at regular intervals, all the frequencies in the model can be scaled by a small constant, which would implement an exponential decay function [How93]. Or, the same process could be carried on locally, on a per-state basis, when the state's total frequency exceeded a threshold [Mof90].

However, regardless of whatever merit direct techniques for recency-weighting stored frequencies may have (none has been shown to consistently improve predictions of blended techniques), these approaches each add an additional feature to the model. Even without such an added feature, every on-line model must by default implement

an inheritance evaluation time; and, the selection of an inheritance evaluation time should be made with some consideration of its appropriateness for the input data. In fact, we can compare the relative effect that different inheritance evaluation times have on model inferences by means of an analogy to family traditions for passing parental knowledge and experience down to children.

Some Intuition

Suppose instead of stochastic models, our suffix-trees represent family trees, where the nodes correspond to people (for simplicity, consider family members of only one sex), the events correspond to situations that the people may find themselves in (such as handling a bully, training a puppy, initiating courtship, buying a used car), and the suffix relationship corresponds to the parent relationship. Here, the "inheritance" received by children is parental knowledge, which is based upon the parent's past experiences plus the parent's own inheritance from the grandparent. Each inheritance evaluation time corresponds to a family tradition for passing knowledge down to children that is strictly maintained by the descendants in each family tree. Note that in this analogy, as in suffix-tree stochastic models, the *weight* that a child assigns to the advice received (inherited) from his parent, relative to his own experience, is a matter completely independent from *when* he receives advice.

The most conservative family tradition allows each parent to pass down only what was passed to him from his parent. This results in each child receiving only the ancient laws that trace back to the family's progenitor. In this tradition, children cannot benefit from their parent's (or grandparent's, ...) experience, and therefore must learn mostly from their own experience. Clearly younger children born to such traditions would have trouble competing with same-age children from more communicative families, although this competitive difference diminishes among older, more experienced children. This corresponds to *inherit at model creation.*

In contrast, the most liberal family tradition requires that every child, before handling any event in his lifetime, regardless of his own experience, listen to advice from his parent, based upon the parent's and other ancestor's current knowledge. The drawback of this approach is frustratingly high communication overhead. However, the benefit is that a parent is able to completely revise the poor advice on a given subject he may have given when he himself was relatively inexperienced. This

corresponds to *inherit at every event visit.*

In the middle lies the approach that is taken naturally by most actual families: children consult parents when they face a completely novel situation, and rely increasingly on their own experience with each recurrence of the situation. This corresponds to *inherit at novel event occurrence.* Blending is a simplification of this approach: children consult parents for novel events but rely thereafter upon their own experience with each recurrence.

Last is the rather anxious approach in which the parent coaches each child on how to handle every imaginable situation as soon as the child is able to record the information, rather than as the situations occur. Assuming a finite number of situations and that the child has perfect memory, there remain two problems with this approach. First, the parent is constantly learning and revising his own knowledge about each of these situations. The later he passes down his knowledge, the higher-quality the advice. In this tradition, children born to inexperienced parents suffer, compared to children born as the parents grow more mature, and they suffer far more than they would if the parents were allowed to revise their early advice. Second, when there are a lot of possible situations, the cost of communicating and remembering them is high, and most of the information will never be of use to the child. This corresponds to *inherit at state creation* and the number of "life's situations" correspond to the size of the input alphabet.

6.4 Computing the Probability Estimate

Once a mixture weighting function $W(s)$ and an inheritance evaluation time have been decided upon, how do we express these decisions in an on-line estimation of the probability of a sequence? In on-line probability estimation, we must compute the sum of the likelihoods $P_e(a_i|\hat{s_i})$ for each a_i in $a_1 a_2 \cdots a_n$, where $\hat{s_i}$ is a state that is specially *selected* as the starting node for the recursive mixture computation using the states excited by $a_1 a_2 \cdots a_{i-1}$. (State selection is the topic of the companion paper [Bun97a].) For the present discussion, assume that $\hat{s_i}$ is the maximum-order excited state at time i). If we are performing (arithmetic) coding or decoding, we must also compute the sum of the conditional probabilities, $P_e(b|\hat{s_i})$, of each b preceding a_i in the (arbitrarily) ordered list of events $b|\hat{s_i}$.

Recall that computing $P_e(b|\hat{s_i})$ requires access to the ancestor likelihood

$P_e(b|\mathbf{suffix}(\hat{s}_i))$. Now, for inheritance evaluation times other than *at every visit*, by definition, for each event $b|\hat{s}_i$, we will not recompute the ancestor likelihood $P_e(b|\mathbf{suffix}(\hat{s}_i))$ for every b that precedes a_i in $\hat{s}_i$'s (the selected state's) event list. Nonetheless, for all states s it is necessary that $\sum_{a \in A} P_e(a|\mathbf{suffix}(s)) \leq 1$, regardless of when the individual $P_e(a|\mathbf{suffix}(s))$ are computed. Ideally we want that sum to be as close to 1 as possible, otherwise, codespace is wasted. The problem is that we cannot count on the current ancestor likelihoods of already-seen events to equal or exceed the ancestor likelihoods we computed for them earlier, since those events may have been seen arbitrarily many times since their likelihoods were computed. Solutions to this problem generally

- under-estimate ancestor likelihoods of veteran events so that their ancestor likelihoods are guaranteed to be less than what they would be if they were recomputed, and
- over-estimate ancestor likelihoods of novel events as they occur to reclaim the ancestor codespace that is wasted by underestimating ancestor likelihoods of veteran events.

6.4.1 Exclusion

We can reclaim the codespace wasted by over-estimated novel-event likelihoods by subtracting the proportion of the ancestor codespace that corresponds to veteran events, before computing a novel event's proportion of the ancestor codespace. This is best accomplished with a technique known as *exclusion* (not to be confused with update exclusion), which was developed for PPM [Mof90]. The basic idea is this: when *exclusion* is enabled for the model, only consider the frequencies of event $a|\mathbf{suffix}(s)$ if the higher-order descendant s is currently excited and event $a|s$ has not occurred before. More precisely, redefine $\mathbf{count} : S \to R$ to be the sum of the event counts for all unexcluded events that have occurred previously following a given state, where a symbol a is defined to be *excluded* at state s if s has an excited child s' such that $\mathbf{count}[a, s', u(s)] > 0$. That is,

$$\mathbf{count}(s) = \sum_{\substack{a\,:\ a \text{ is not excluded} \\ \mathbf{count}[a,s,u(s)] > 0}} \mathbf{count}(a, s).$$

Unless stated otherwise, we shall assume that exclusions are enabled in all computations described from here on.

6.4.2 Blending's Missing Term

Blending [CW84b] evaluates the ancestor likelihood $P_e(a|\mathbf{suffix}(s))$ *before novel event updates*, but at all subsequent occurrences of any string in the set $L(s) \cdot a$. Blending assumes that $P_e(a|\mathbf{suffix}(s)) = 0$, and thereby drops a term of our mixture formula, for events that are not novel. This certainly ensures that the reused ancestor likelihoods for veteran events are less than they would be if they were recomputed from the current frequencies at s's ancestors. The result is that subsequently computed probability estimates of the veteran event $a|s$ are slightly deflated, while exclusions ensure that future estimates of all symbols that have yet to be seen following the a member of the context $L(s)$ will be slightly inflated.

6.4.3 State Variables for Mixture Computation

In general, the inheritance evaluation time *before novel event updates* makes it difficult to ensure that

$$\sum_{a:\mathbf{count}[a,s,u(s)]>0} P_e(a|\mathbf{suffix}(s)) + \sum_{a:\mathbf{count}[a,s,u(s)]=0} P_e(a|\mathbf{suffix}(s)) \leq 1.$$

However, an alternative is to satisfy the requirement that

$$(1-W(s))\cdot\left(\sum_{a:\mathbf{count}[a,s,u(s)]>0} P_e(a|\mathbf{suffix}(s)) + \sum_{a:\mathbf{count}[a,s,u(s)]=0} P_e(a|\mathbf{suffix}(s))\right) \leq (1-W(s)),$$

which can be accomplished less drastically than blending's solution of setting the ancestor likelihood $P_e(a|\mathbf{suffix}(s))$ to zero for any veteran event $a|s$. Instead, we subtract $(1 - W(s)) \cdot P_e(a|\mathbf{suffix}(s))$ from the ancestor code space $(1 - W(s))$ after the ancestor likelihood is first computed. This is accomplished by implementing the mixture formula using the additional state-variables $\alpha[s]$, which replace $\alpha(s)/d(s)$; and $I[a, s]$, which is required for computing $I(a, s)$ when inheritance evaluation time equals *before novel event updates* (although we use use it for all evaluation times in our cross-product implementation). The other required state variables are the event frequencies, $\mathbf{count}[a, s, 0]$ and $\mathbf{count}[a, s, 1]$, which are required for correctly

combining state selection with either type of update exclusion (i.e., *regular update exclusion* or *maximum-order updates*); plus an exclusion vector, *Excluded*$[a]$, which records which symbols were excluded by higher order excited nodes, and which must be reset for each input sequence symbol a_i.

Initially, $\mathbf{count}[a, s, 1] = \mathbf{count}[a, s, 0] = 0$, and $\alpha[s] = z(s)$, where $z(s) = 1$ if weighting function 'A' is used, and $z(s) = 0$ for the mixture variants discussed so far.[1] Then, for each input symbol a_i, after a_i's probability has been computed and its codepoint transmitted, the frequencies corresponding to a_i must be updated at all states excited by $a_1 a_2 \cdots a_{i-1}$ as follows:

$$\forall s : s \in S, a_1 a_2 \cdots a_{i-1} \in L_i(s),$$

$$\alpha[s] = \begin{cases} \alpha[s] + 1 & \text{if } a|s \text{ is novel and WeightFunction} \neq \text{'A'}, \\ \alpha[s] & \text{otherwise.} \end{cases}$$

$$\mathbf{count}[a, s, 1] = \begin{cases} \mathbf{count}[a, s, 1] + 1 & \text{if } s \text{ has no excited children;} \\ \mathbf{count}[a, s, 1] + 1 & \text{if MaxOrderUpdates = FALSE and } a|s \text{ or } a|s' \text{ is novel, where } s = \mathbf{suffix}(s'); \\ \mathbf{count}[a, s, 1] & \text{otherwise.} \end{cases}$$

$$\mathbf{count}[a, s, 0] = \mathbf{count}[a, s, 0] + 1.$$

The value of $I[a, s]$ is determined during probability estimation, for each excited state s that equals the selected state $\hat{s}_i$ or one of its ancestors. In a compressor or decompressor, probability estimation is intertwined with arithmetic coding. The relationship between arithmetic (de)coding and probability estimation that computes mixtures of suffix-tree frequency distributions is described in the recursive procedure *code*: $S \times A \times \{0, 1\} \rightarrow [0 \ldots 1)$ given in Figure 6.1.

The procedure in Figure 6.1 shows how the set of inheritance times affect the recursive mixture computation when applied to on-line coding. However, the pseudocode is designed for generality without obfuscating the conceptual simplicity: any actual implementation (including ours) must differ to be more computationally efficient and to avoid the instabilities of floating-point arithmetic. One optimization in

[1] For DMC variants, $z(s)$ will be initialized to the frequency of the edge redirected when s was added to the model.

```
procedure code(s ∈ S, a ∈ A ∪ {λ},  Coding ∈ {0,1})
        r, x, sum, I_s ∈ ℜ;   b ∈ A;
        sum ← 0.0;   r ← numerator(W(s))/count(s);   I_s ← Σ_{c:Excluded[c]} I[c,s];
        repeat  b ← symbol of next unexcluded event in s's event list;
                if Exclusion then Excluded[b] ← True endif
                if Inherit_Time = At_Every_Event_Visit then
                        ∀p ∈ ancestors(s) ∪ {s},  I[b,p] ← P_e(b|suffix(p)) · α[p]
                endif
                I_s ← I_s + I[b,s];   x ← r · count(b,s) + I[b,s];
                if Coding then
                        if b ≠ a then sum ← sum + x
                        else         arith_renorm_transmit(sum, x, denominator(W(s)))
                        endif
                else
                        if sum + x < Codepoint · denominator(W(s)) then sum ← sum + x
                        else         a ← b;   output a;
                                     arith_renorm_receive(sum, x, denominator(W(s)), Codepoint)
                        endif
                endif
        until b = a or s's event list is exhausted;
        if b ≠ a then
                if Coding then arith_renorm_transmit(sum, α[s] − I_s, denominator(W(s)));
                else     arith_renorm_receive(sum, α[s] − I_s, denominator(W(s)), Codepoint);
                endif
                Insert novel event a|s into s's event list;
                I[a,s] ← (α[s] − I_s) · code(a, suffix(s));   x ← I[a,s]
        else if Exclusion then ∀b|s ∈ s's event list, Excluded[b] ← False endif
        return(x/denominator(W(s)))
end procedure
```

Figure 6.1: On-line (de)coding of event $a = a_i$ using recursive *mixture* with *inheritance*.

particular bears mention: with any of the escape mechanisms 'A', 'B', 'C', or 'D', and *inherit before novel event update*, $I[a, s]$ can be permanently subtracted from $\alpha[s]$ if **count**() is redefined to be $\mathbf{count}(a, s) = \mathbf{count}[a, s, u(s)] + k + I[a, s]$.

During compression, *code*($\hat{s}_i, a_i$,1) computes and returns the probability of the currently scanned sequence symbol a_i and incrementally transmits the high-order bits of the decoder's global variable *Codepoint*, which identifies the unique subinterval of $[0 \ldots 1)$ that corresponds to a_i, using an arithmetic coder. During decompression, *code*($\hat{s}_i, \lambda, 0$) incrementally receives the high-order bits of the *Codepoint*, and computes and returns the probability of the symbol, a_i, that corresponds to the unique subinterval of $[0 \ldots 1)$ that contains the value of *Codepoint*. (The probability of a_i equals the width of a_i's subinterval in $[0 \ldots 1)$.) The procedures *arith_renorm_transmit()* and *arith_renorm_receive()* manage the arithmetic coder's and decoder's internal states, respectively, including the decoder's *Codepoint*. The *Codepoint* is incrementally transmitted (high-order bits first) using the subinterval endpoints that are specified at the low end by

$$sum/_{\text{denominator}}(W(s)),$$

and at the high end by

$$(sum + {}_{\text{numerator}}(W(s)) \cdot \mathbf{count}(a_i, s) + I(a_i, s))/_{\text{denominator}}(W(s))$$

if event $a_i|s$ is not novel, or

$$(sum + \alpha[s])/_{\text{denominator}}(W(s))$$

if $a_i|s$ is novel.

6.5 Probability Estimation in DMC

The remaining undescribed aspect of DMC and the remaining undescribed features of our mixture technique both involve how DMC computes its probability estimate. DMC can be viewed as computing a mixture of the form we have described; however, DMC weights the mixture differently from any PPM variant and it permanently alters the ancestor-state sources of inherited frequencies as well. Recall that DMC adds each new state to the model as the new destination of a redirected transition.

When a new state is first added to a DMC model, the *observed* frequencies of all symbols given that state are zero, and therefore cannot be the basis of a probability estimate. So, DMC computes a complete initial frequency distribution for each newly added state when it is created, using the frequency data at the original destination of the redirected transition. We showed in Chapter 4 that the original destination state is the **suffix** of the new state. Thus, DMC's approach is very similar to computing a mixture with inheritance evaluation time set to *at state creation*. The principal difference is that DMC subtracts the frequency distribution assigned to the newly added child state from its parent's distribution, and initializes its mixture weights to permit this subtraction. That is, DMC permanently subtracts the inheritance at a new state s from the frequency distribution at its parent, $\mathbf{suffix}(s)$.

6.5.1 Frequency Distributions and Cloning in DMC

In DMC's initial model, M_0, a uniform frequency distribution is assumed over the (reflexive) out-transitions of s_0. Frequency updates can be viewed either as *max-order updates* or as *update exclusions* (Chapter 5), which have equivalent results when combined with *inherit at state creation*. The weighting function 'A' allows emulation of DMC with our mixture formula, because 'A' initializes event frequencies to zero and 'A' prevents $I[a, s]$ from requiring re-evaluation by freezing $\alpha[s]$ at its initial value $z(s) = \mathbf{count}[\mathbf{symbol}(s), \mathbf{prefix}(s), 1]$. Other weighting functions, which increase $\alpha[s]$ as novel events occur, can be combined with mixtures that set $z(s)$ as DMC does and freeze $P_e(a|\mathbf{suffix}(s))$ at the value determined at the time state s was added, but $I[a, s]$ will have to be re-evaluated each visit.

As each s_{k+1} is added, it becomes the new destination of a transition corresponding to the event $c|s_p$, and each out-transition from the original destination, s_t, is copied and assigned an inherited frequency. That is,

$$\forall b \in A,\ I[b, s_{k+1}] \leftarrow \mathbf{ratio} \cdot \mathbf{count}[b, s_t, 1],$$

where $s_t = \mu_k(s_p, c)$ and

$$\mathbf{ratio} \leftarrow \mathbf{count}[c, s_p, 1] / \Sigma_{b \in A} \mathbf{count}[b, s_t, 1].$$

Then, the *update-excluded* frequencies of the copied out-transitions are *reduced* by the frequencies assigned to the copies,

$$\forall b \in A,\ \mathbf{count}[b, s_t, 1] \leftarrow \mathbf{count}[b, s_t, 1] - I[b, s_{k+1}].$$

For all other transitions in the model, event counts and inherited frequencies remain unchanged.

This process is pictured in Figure 6.2, using the the definitions of Section 4.3.1, that is, $s_t = \mathbf{suffix}(s_{k+1})$ and $s_p = \mathbf{prefix}(s_{k+1})$. The process of redirecting an edge to a new state can be viewed as a reclassification of the strings that take FSM M_k across that edge. Note the resulting 'homogeneity assumption' regarding the inherited distribution of next-symbol frequencies that are conditioned by the redirected edge, relative to the original frequency distribution at its original destination $\mathbf{suffix}(s_{k+1})$ that is implicit here. A more exact (and expensive) way to compute the new state's inheritance would require having each edge remember the exact distribution of next-symbol frequencies that are conditioned by strings which have crossed the edge in the past.

6.5.2 Lazy Cloning and other DMC variants

Since cloning corresponds to computing an inheritance *at state creation*, we can postpone computing the inheritance and redirecting the associated transition corresponding to an event until that event occurs for the first time. The result, *lazy cloning*, is pictured in Figure 6.3. The principal difference between lazy cloning and mixtures that inherit before novel event updates is whether or not the inheritance is subtracted from the **suffix** state's frequency distribution. The second difference is that emulating DMC requires adding another implementation parameter to the weighting functions: $z(s)$ is the initial value for the counter $\alpha[s_{k+1}]$, and is set to the frequency of the redirected transition, that is, $z(s) = \mathbf{count}[\mathbf{symbol}(s_{k+1}), \mathbf{prefix}(s_{k+1}), 1]$.

Actually, if we define a *cloning mixture* as any mixture that initializes $\alpha[s]$ to the value of $\mathbf{count}[\mathbf{symbol}(s_{k+1}), \mathbf{prefix}(s_{k+1}), 1]$, we can generalize cloning in several ways. For example, any inheritance time can be combined in a cloning mixture that uses weighting function 'A.' Alternatively, any weighting function can be combined in a cloning mixture that inherits *after* state creation. The choice between whether or not the inheritance is subtracted from the clone's parent distribution constitutes a third option for cloning mixtures, and can be combined with *inherit at state creation* or *inherit before novel event update*.

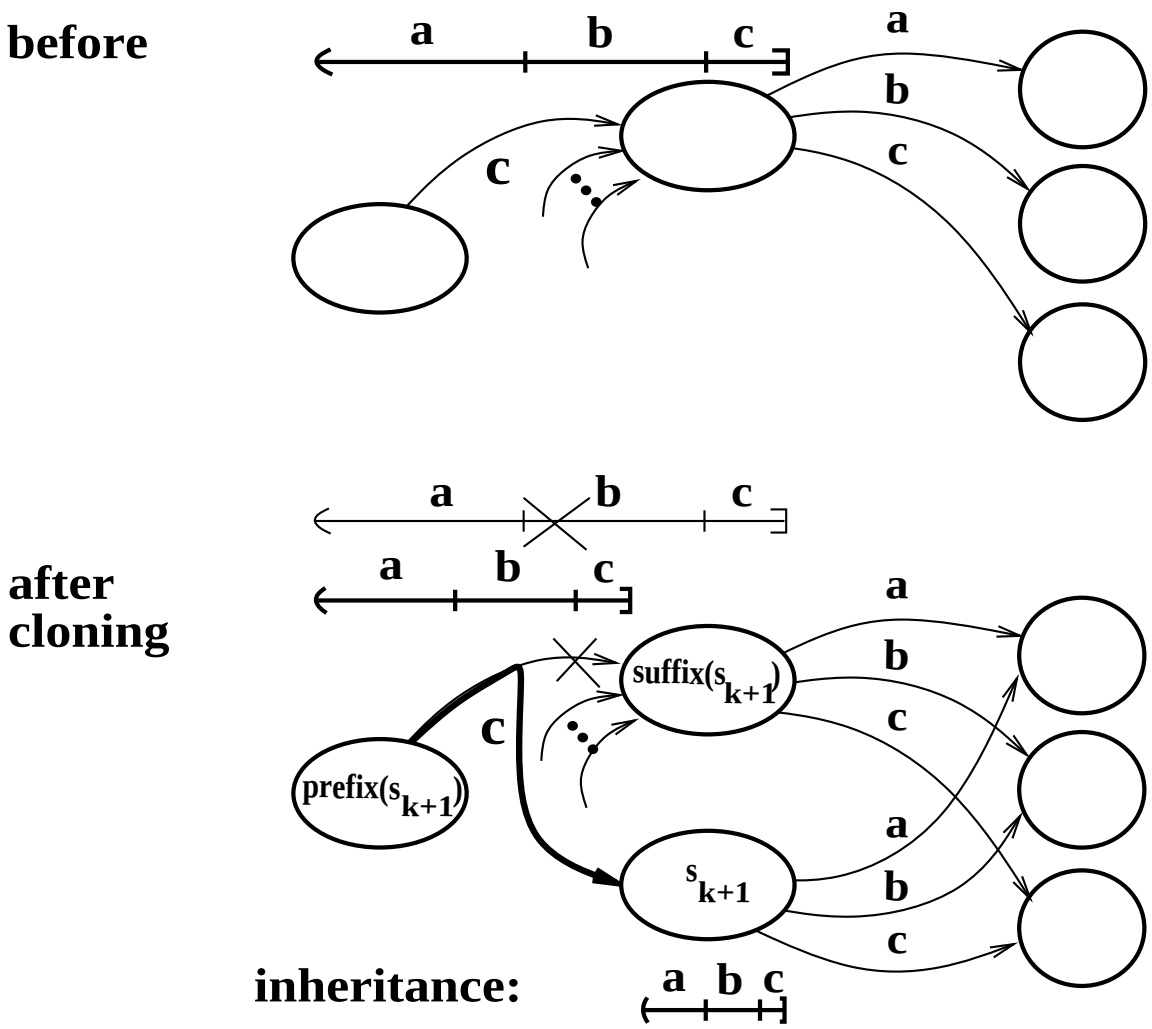

Figure 6.2: **Cloning, Frequency Distributions, and Retractions in DMC.** When an eligible edge redirected, its new destination is a clone of its former destination, and the frequency distribution at the original destination is divided between the two destinations. The frequencies in the clone's copy of the distribution (the *inheritance*) sum to the number of times the redirected edge has been traversed, and the original destination state keeps the remaining frequencies. Note that the number of distribution subinterval calculations and the number of newly added edge pointers corresponds to the size of the input alphabet $|A|$, so in practice, DMC has always taken a binary input alphabet.

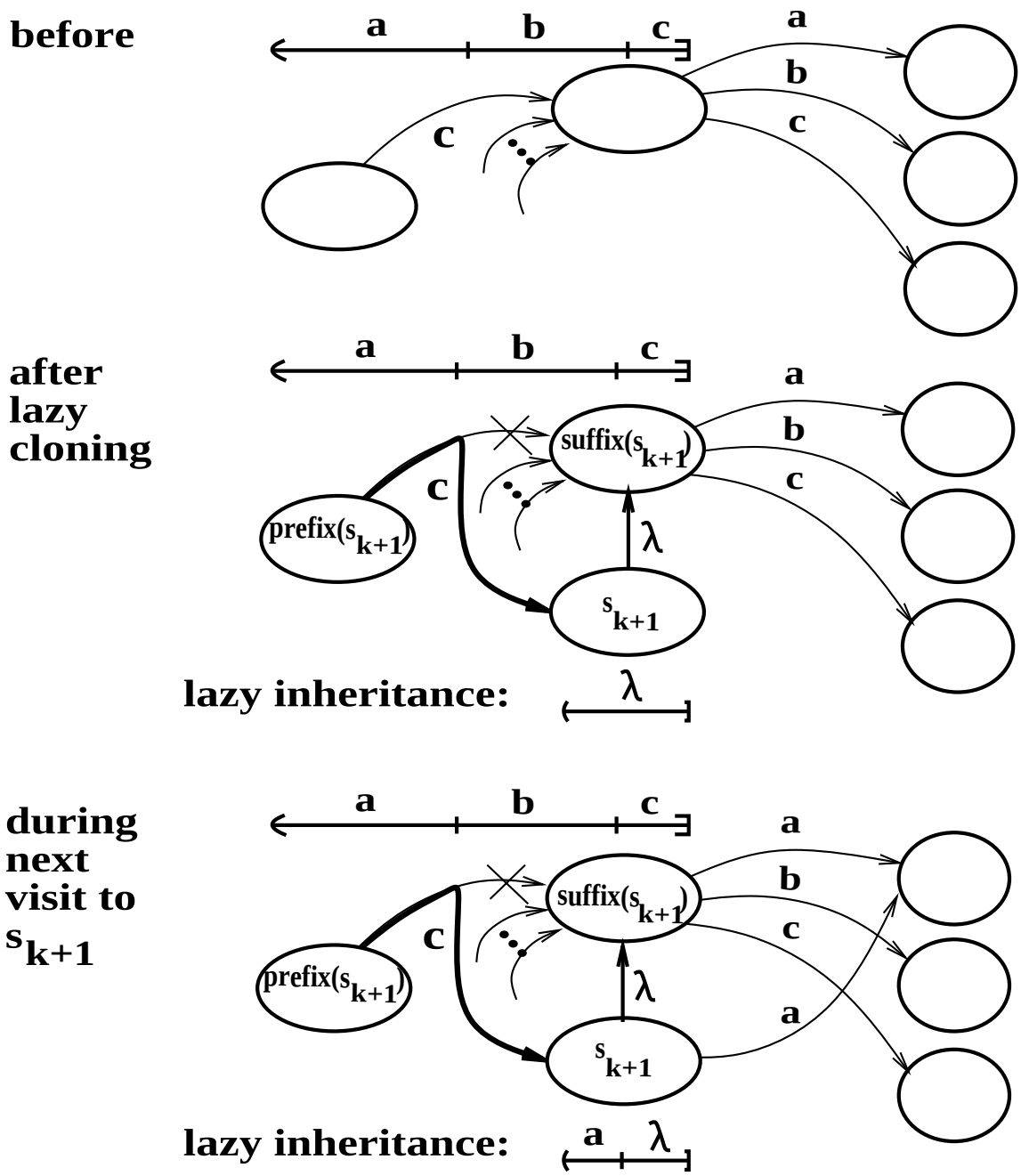

Figure 6.3: **Lazy Cloning in DMC.** When a new state is created, the frequency of the event that a novel symbol is seen (λ) is set to the frequency of the redirected edge, and thus the probability of traversing the **suffix**() pointer when in that state is 1.0. Whenever a novel symbol is seen in a given state, its portion of the state's inheritance is recursively evaluated, before the symbol's probability is estimated and before any frequency updates occur at the given state. Note that a novel symbol's portion of the inheritance is subtracted from λ's frequency and added to the symbol's frequency.

6.5.3 Cloning and PPM

Incidentally, PPM and PPM* can be emulated using cloning. Simply set the edge-redirection criterion so that it will redirect a transition upon creation if its destination has order less than the order bound. For computing mixture weights, any of the functions used with PPM will do. Simply set the parameter $z(s)$ to the frequency of the redirected edge, which is always zero with this cloning criterion.

6.6 Summary

We must always compute some sort of *mixture* when computing on-line probability estimates with suffix-tree FSMs, and a recursive mixture is completely defined in terms of its recursive weighting function $W(s)$, and its *inheritance evaluation time*. For example, PPM's *blending* is a forgetful type of mixture that lazily evaluates its inheritances as novel events occur, while DMC's "cloning" [CH87] produces a mixture that evaluates its inheritances when new states are added, but which also subtracts the inherited frequency from the parent distribution. The weighting functions and inheritance evaluation times are independent of each other and of whether inheritances are subtracted from parent distributions; thus *mixtures* generalize both PPM's *blending* and the quantitative aspects of DMC's *cloning*.

In earlier chapters we presented two model structuring mechanisms (that is, string-transition splitting for PPM variants, and symbol-transition redirection for the structural aspects of DMC's cloning). These mechanisms determine the model's linguistic family and are orthogonal to the three mixture descriptors above. In the previous chapter we presented another independent design option: three ways to organize the frequency data into the frequency distributions that are combined when computing mixtures. This chapter introduced a generalization and orthogonal decomposition of blending into mixture weighting formula, inheritance evaluation time, initial inheritance mass, and whether inheritances are subtracted from parent frequency distributions. In the next chapter, we complete our basic list of orthogonal sets of design options for describing suffix-tree models by adding a set of techniques that perform information-theoretic model selection. Afterwards, we give controlled empirical studies of various elements of the cross product of these orthogonal sets, which were executed using an implementation of that cross product.

Chapter 7

SELECTING THE CODING MODEL

This chapter introduces into practice a set of techniques for *information-theoretic state selection* that have been developing in asymptotic results for over a decade. State selection, which actually implements the selection of a model from among a set of competing models, is performed at least trivially by all suffix-tree FSMs used for on-line probability estimation. The set of state-selection techniques presented in this chapter combine orthogonally with the other sets of design options covered so far.

7.1 Stochastic Complexity

The *stochastic complexity* of a string is the length of its optimal off-line encoding, that is, its Minimum Description Length, or MDL [Ris89]. A string's MDL is the sum of the lengths of an encoding of a model plus the encoding of the string with respect to that model such that the total encoding length is minimal over all possible models within an assumed model class. Here we consider the problem of computing the input sequence's MDL on-line, by assuming for each input symbol that the set of possible FSM models is represented by the set of nested subtrees of the suffix-tree FSM. Our goal is to code each input symbol with the model that assigns the lowest stochastic complexity to the already processed portion of the input. At any point in the input sequence, each nested FSM is in a particular state. The set of current states of the nested FSMs is the set of *excited* suffix-tree states. By selecting one of the excited states, we are in fact selecting an entire FSM with that state as its current state.

Note that stochastic complexity assigns a coding penalty to each model state. The penalty is a lower bound on the number of bits required to encode that state. Most other treatments of the state-selection approach to on-line modeling (e.g., [Ris83, WLZ92]) require that a refinement (e.g., the children or deeper descendants) of a state be selected in preference to that state if the performance of that refinement improves the performance of the model frontier containing the original state by an amount

exceeding the cost of encoding the states comprising the refinement. However, for the following reasons, we believe that there should be no coding penalty for selecting a refinement to a given state during *on-line* modeling:

- In on-line modeling, the model is not explicitly coded: the deterministic algorithm of the encoder is emulated by the decoder to deduce the state that was selected for coding without any side-information about the actual model.
- In on-line modeling, the coding penalty is incorporated into the inaccurate probability estimates from early in the sequence [CW84a]. Thus, refinement coding penalties are incorporated into the records of past performance based on those estimates.
- Automated experiments with various parameterizations of our executable taxonomy, which evaluate the effect of loose constant lower bounds on the off-line coding cost of model refinements, indicate that positive coding penalties degrade on-line performance.

Therefore, for each input symbol a_i, we shall simply select the excited state that represents the model that has *performed* the best on the input sequence $a_1 a_2 \cdots a_{i-1}$.

7.2 A Performance Metric for States

Thus we need to associate a *performance metric* with each state to use for minimizing the codelength of the entire sequence $a_1 a_2 \cdots a_n$. The optimal codelength equals

$$\sum_i -\log P(a_i | a_1 a_2 \cdots a_{i-1}),$$

and we model all past occurrences of all suffixes of $a_1 a_2 \cdots a_{i-1}$ using the currently *excited* states $\{s : a_1 a_2 \cdots a_{i-1} \in L(s)\}$. We therefore wish to minimize the minimal codelength that will be assigned by the probability estimator to symbol a_i given the currently excited states,

$$-\log P_{\text{estimated}}(a_i | \{s : a_1 a_2 \cdots a_{i-1} \in L(s)\}),$$

for all a_i, where $L(s)$ is the set of strings that cause the nested FSM represented by the suffix-tree to enter state s or any state in the subtree rooted by s.

A performance metric that will help accomplish this goal is maintained as follows: For each state s we maintain a counter called $D[s]$ that accumulates the codelengths

that state s would have assigned to the symbols that were currently scanned whenever it has been excited. That is, for each input a_i, at all s such that $a_1 a_2 \cdots a_{i-1} \in A^* L(s)$, we increment $D[s]$ by $-\log P_e(a_i|s)$. We also keep track of $\sharp s$, the number of times that s has been excited since it was added to the state set. Then, we measure the performance of each state s by its expected codelength, $D[s]/\sharp s$. From here on we shall loosely refer to each state's expected codelength $D[s]$ as its "MDL." Now we are prepared for selecting, for the purpose of assigning a probability to each successive source symbol, the best-performing model represented by the excited states.

7.3 Basic Approaches to State Selection

There are three basic approaches to information-theoretic state selection. Each approach can be viewed as selecting a *complete frontier* of the subtree rooted at s_0, where a frontier of state s's subtree T consists of the leaves of some subtree of T rooted at s, and a frontier is *complete* if it consists of s or of complete frontiers of all of s's children.

1. Top-down from s_0, select the first state s whose children's combined expected codelengths fail to improve s's expected codelength.

2. Bottom-up from the maximum order excited state, select the excited child of the first state s whose children's combined expected codelengths improve s's expected codelength.

3. Top-down, select the minimum order excited state for which no complete frontier of its subtree improves its expected codelength.

The first approach was introduced, using entropies instead of MDLs [Ris83], and then later using MDLs [Ris86a]. This technique systematically under-estimates the local order of the model [WLZ92]; it is a hill-climbing technique that can get stuck in local minima. The second approach is introduced here as an obvious complement to the top-down hill-climbing approach, to complete the taxonomy. It systematically over-estimates local order. Both methods are most efficiently implemented using a single MDL counter $D'[s]$ at each state s, which records the difference between the per-symbol codelength assigned by the state s and the codelength assigned by the currently excited child of s. That is, given that $D'[s] = 0 \ \forall s \in S$ initially, for every

state s such that s has an excited child t at time i, let

$$D'[s] \leftarrow D'[s] - \log(P_e(a_i|s)) + \log(P_e(a_i|t)).$$

Both hill-climbing methods approximate the third approach. The reason that the hill-climbing approaches are suboptimal is that there may be a complete frontier below any given state's children that reduces the state's expected codelength, even though the children themselves do not. Weinberger, *et al* present a formal, asymptotically convergent solution to the third approach that requires an order bound [WLZ92]. Below, we describe our own solution, which requires no order bound and which allows efficient implementation. But first we will explain certain semantic considerations involving context partitions and frequency updates.

7.4 Model Semantics III: Competing Context Partitions

State selection in a suffix-tree FSM implements the selection of an entire partition on the set of possible conditioning contexts, where the partition element associated with each state is the set of strings that will cause the FSM to enter that state or any state in its subtree. We restrict ourselves to the selection of complete frontiers because only complete frontiers impose a complete partition on the set of conditioning contexts. For every possible input history $a_1 a_2 \cdots a_{i-1}$, there must exist a state in the coding model selected at time i whose conditioning context contains $a_1 a_2 \cdots a_{i-1}$.

It is fortunate that a state-selection procedure need only consider the metrics located at the excited states. However, the designer must remember that he or she is actually implementing the selection of an entire conditioning context partition from among a set of competing partitions. This means that a node must always be considered for selection *along with its siblings*. To select, for example, the excited state with the lowest expected codelength, or the minimum-order excited state whose expected codelength is better than that of its excited child [Fur91], is incorrect, not merely suboptimal. This is because the children of a state s (i.e., those nodes whose contexts correspond to minimal extensions of the state's context) may have better performance than the state s, even while the currently excited child of s has worse performance than s does.

Frequency updates affect the conditioning context partition imposed by the state set. Therefore, the choice of update mechanism must be carefully considered in com-

bination with state selection. At first glance, only *full updates*, which increment the frequency distribution at every excited state (and therefore at every simulated FSM model), seem appropriate for models with state-selection. However, there are two reasons to combine state-selection with *update exclusion*, which increments frequencies at only the highest-order excited states. First, update exclusion improves performance of mixtures (which we propose to combine with state selection). Second, update exclusion correctly handles the *incomplete* frontiers that result from lazily evaluating refinements to suffix-tree states [Bun96, see Section 7.6]. With mixtures, the probability estimate is defined recursively in terms of ancestor nodes. State selection does not cause us to assume that the children of these lower-order nodes do not exist, therefore update-excluded frequencies should be used to compute the lower-order terms of the probability estimate, even when state selection is employed. Regardless of whether the model uses mixtures or constructs incomplete frontiers, update-excluded frequencies must not be used to compute the probability estimate at the selected state, since the act of selecting a state assumes that the descendants of the selected state do not exist. Thus, disabling update exclusion at the currently selected state is required when it is enabled globally for the modeling algorithm.

To allow update-exclusion to be selected on a per-state basis when it is globally enabled for the modeling algorithm, we have introduced a dual update mechanism (see Chapter 5) that maintains both update-excluded and full-update frequencies at every state.

7.5 A Percolating State-Selection Mechanism

Here we present a dynamic-programming solution to the problem of finding the best-performing model frontier without resorting to hill-climbing or an order-bound. In simple terms, our solution recursively "percolates" the performance of each subtree's best frontier up to its root's ancestor. Figure 7.1 gives the two procedures, *Percolate_MDLs*: $S \times A \rightarrow \{\}$ and *Select*: $S \rightarrow S \cup \{null\}$ that implement the two principal steps. These procedures require two MDL accumulators at each actual state s, $D[s]$ and $F[s]$, which respectively contain the state's locally accumulated MDL, and the accumulated MDL of the best complete frontier in the subtree rooted by the state.

Select(s') follows suffix pointers from the current maximum-order excited state s' to the root s_0, and then searches top-down for the first excited state s such that

```
procedure Select (s ∈ S)
    selected ∈ S ∪ {null};
    if s ≠ s_0 then
        selected ← Select(suffix(s));
        if selected then return selected endif
    endif
    if D[s] ≤ F[s] then return s else return null endif
end procedure


procedure Percolate_MDLs(s ∈ S, a ∈ A)
    old_D, old_F ∈ ℜ;
    old_F ← F[s];
    F[s] ← F[s] − log(P_e(a|s, update_exclusion = true));
    while s do
        old_D ← D[s];
        D[s] ← D[s] − log(P_e(a|s, update_exclusion = false));
        if suffix(s) then
            diff ∈ ℜ;
            diff ← min{D[s], F[s]} − min{old_F,old_D};
            old_F ← F[suffix(s)];
            F[suffix(s)] ← F[suffix(s)] − diff
        endif
        s ← suffix(s)
    end while
end procedure
```

Figure 7.1: A State Selection Mechanism with Percolating Updates

$D[s]$ is no greater than $F[s]$. For each excited state s, $Percolate(s, a_i)$ recomputes, bottom-up, the local description length, $D[s]$, and $F[s]$, the description length of the best-performing frontier in s's subtree. The procedures in Figure 7.1 implement the following bottom-up recomputation of $F[s]$ for each state s, starting from the maximum-order excited state s', by resetting $F[\mathbf{suffix}(s)]$:

$$F[\mathbf{suffix}(s)] \leftarrow$$
$$F[\mathbf{suffix}(s)] - \min\{D[s], F[s]\} + \min\{D[s] - \log P_e(a_i|s), F[s] - \log P_e(a_i|s_j)\},$$

where s_j is the excited state located on the best-performing frontier of the subtree

rooted by s.

For on-line suffix-tree modeling algorithms in general, calls to *Select* and *Percolate_MDLs* fit into the sequence of calls to the other routines as follows. At the beginning of processing the sequence $a_1 a_2 \cdots a_n$, the model consists of the (excited) state s_0, and its parent, s_{-1}, where $F[s_0] = D[s_0] = F[s_{-1}] = D[s_{-1}] = 0$. Before each subsequent symbol a_i is processed by the probability estimation routines, *Select* is called to select the best excited state $\hat{s}$. Then, $P(a_i|a_1 a_2 \cdots a_{i-1})$ is estimated as $P_e(a_i|\hat{s})$. After a_i has been processed by the probability estimation routines, the maximum-order excited state s' is tested for eligibility to be extended by new states. If any new states s are added to the model, they are added top-down as suffix-linked descendants of s', before *Percolate_MDLs*() is called. The MDL counters at new states are initially zero, that is, $F[s] = D[s] = 0$, and the pointer to s' is set to the maximum-order novel state. Then, the MDL counters are updated by executing *Percolate_MDLs*(s', a_i). Finally, the event frequencies are updated at all excited states before advancing to the next scanned sequence symbol a_{i+1} and setting s' to the maximum-order state that is excited by $a_1 a_2 \cdots a_i$.

7.6 Model Semantics IV: Incomplete Frontiers

Most implementations of data compression algorithms, including the methods studied in this work, add children states *on demand* instead of all at once. Thus, few states in the suffix-tree will posess all possible children. Here, we explain how to use update-excluded frequencies during MDL updates to ensure that the MDLs keep track of the performance of models corresponding to *complete* context partitions, even in incomplete suffix trees.

In an on-line suffix-tree FSM in which a state s does not posess all possible children, the refinement to s's context partition element $L(s)$ that is represented by s's children is incomplete because

$$L(s) - \bigcup_{t:\mathbf{suffix}(t)=s} L(t) \neq \{\}.$$

To complete s's context partition element, we maintain a "shadow child" of s that

maintains a next-symbol frequency distribution that is conditioned by

$$L(s) - \bigcup_{t:\mathbf{suffix}(t)=s} L(t).$$

As explained in [Bun96], Chapter 5, this is the same frequency distribution that is formed by the update-excluded frequency counts $\mathbf{count}[-, s, 1]$ if maximum-order updates are globally enabled for the modeling algorithm. Alternatively, it is approximated very closely by the update-excluded frequency counts $\mathbf{count}[-, s, 1]$ if regular update exclusion is globally enabled instead. *Therefore, update-excluded frequencies, in addition to full-update frequencies, are required to correctly implement state selection in context models that lazily refine their context partitions, independently of whether update-excluded frequencies are used when estimating the coding distribution.*

Whenever no state is selected from among the currently excited states, the "shadow child" of the maximum order excited state s', is the selected state, and the local order estimate equals the order of s' plus one. To implement the selection of a complete frontier, the probability estimator should use the update-excluded frequencies conditioned by s' when computing the coding distribution, and when updating the MDL $F[s']$ of the best-performing frontier below that state. That is, during MDL updates, increment $F[s']$ by $-\log P_e(a_i|s', \mathit{update\_exclusion} = \mathit{true})$, which equals the shadow child's codelength, and update $D[s']$ by the maximum order excited state's codelength, $-\log P_e(a_i|s', \mathit{update\_exclusion} = \mathit{false})$. The quantity $P_e(a_i|s', \mathit{update\_exclusion} = u)$ equals a weighted sum of the *inheritance* $I[a_i|s']$, and the maximum likelihood of a_i given the frequencies $\mathbf{count}[b, s', u]$, $\forall b \in A$, as computed in Chapter 6.

7.6.1 Implementation Issues

When PPM* or PPM are implemented with string-transitions, initial values of $F[\,]$ and $D[\,]$ are intimately bound to the presence of virtual states that lie between the symbols of a string that labels a transition. All states except s_0 and s_{-1} exist as virtual states before they become actual states. Virtual states require no MDLs: if no actual node is selected, then the minimum order (i.e., lowest entropy) excited virtual node v is selected. In that case, $P_e(a_i|a_1a_2\cdots a_{i-1})$ is computed as $P_e(a_i|v)$ using no update exclusion at v. If no actual node is selected and there are no excited virtual states, $P_e(a_i|a_1a_2\cdots a_{i-1})$ is computed as $P_e(a_i|s')$ using no update exclusion.

Any time the excited virtual states are not split, the minimum-order virtual state's codelength is added to $F[s']$ at the maximum excited actual state s'. Then, assuming the mixture weighting function is not 'B', when an excited virtual state v is split into a new actual node s, the $D[s]$ and $F[s]$ are both initialized to

$$-\log\left(P_e(a_v|s')\cdot\prod_{1\le i<\sharp v}\frac{i}{i+\alpha}\right),$$

where

- a_v is the symbol that followed v in the past,
- $P_e(a_v|s')$ is the probability estimate of a_v at the existing maximum order excited actual state s' (computed with update exclusions enabled for s'),
- $\sharp v$ is the number of times that state v has been excited in the past, and
- α equals .5 with mixture weighting scheme 'D', and $\alpha = 1$ if scheme 'A' or 'C' is used.

If the mixture weighting function is 'B', $D[s]$ and $F[s]$ are both initialized to

$$\sharp v\cdot -\log P_e(a_v|s'),$$

if $\sharp v < 2$, and

$$2\cdot -\log\left(P_e(a_v|s')\cdot\prod_{2\le i<\sharp v}\frac{i-1}{i}\right),$$

otherwise.

Deducing the frequencies is straightforward; however, deducing the true accumulated codelengths would require child and sibling pointers at the source of the string transition, plus significant searching overhead. Thus the procedure given above *approximates* the MDLs of virtual nodes when they are split, by imposing the following simplifying assumptions:

- A virtual state can have no (virtual) children other than its excited child.
- The excited virtual states were added to the model simultaneously; thus s' provided each virtual state's initial probability estimate.
- The initial estimate $P_e(a_v|s')$ has not changed since the virtual states' context strings first occurred.

The effects of these approximations are measured against an exact implementation in Section 9.

7.7 State Selection with Mixtures

In general, an on-line modeling algorithm that uses convergent state selection will ignore the high-order descendants of a given node until the combined estimated probability distributions of those descendants have better performance, or lower entropy, than the frequency distribution at the given node. A simple explanation of why blending and other mixtures work so well in on-line modeling algorithms is that these techniques enable a given state's probability estimate to converge to the characteristics of the input data sooner. Accelerating this convergence is essential to a state's usefulness if it has high Markov order. Even for large input sequences, high-order states are invariably starved for data. In practice, mixtures accelerate the convergence at particular *states*. That is, mixtures lower the expected entropy of their estimated probability distributions.

Thus, the combination of mixtures with state selection will accelerate the convergence of *models*. That is, higher-order states will be selected sooner than with state selection alone. The algorithm resulting from this combination should produce a model that performs well for both short and long sequences.

7.8 Summary

Before this thesis was conceived, there had been no published empirical studies of the performance of any state-selection technique, nor was the idea used in any published implementation. In this chapter we explained the concepts and existing state-selection techniques from the information-theoretic literature, and then presented a novel technique that overcomes the drawbacks of the existing techniques, which resort to order-bounds or suboptimal hill-climbing. With this chapter we have completed the description of the principal orthogonal elements of a cross-product taxonomy for on-line suffix-tree models of sequences. In the next chapters, we use our implementation of that cross product to present controlled experiments that conclusively demonstrate our main hypothesis: that the combination of information-theoretic state selection and mixtures delivers performance superior to that of either technique alone.

Chapter 8

AN EXECUTABLE TAXONOMY OF ON-LINE MODELING ALGORITHMS

This chapter gives an overview of our decomposition of a group of existing and novel on-line sequence modeling algorithms into component parts. Our decomposition, and its implementation, show that these algorithms can be implemented as a cross product of predominantly independent sets of algorithmic features. The result is all of the following: a *test bed* for executing controlled experiments with algorithm components, a *framework* that unifies existing techniques and defines novel techniques, and a *taxonomy* for describing on-line sequence modeling algorithms precisely and completely in a way that enables meaningful comparison.

8.1 Design Philosophy

The executable cross product and nomenclature described here are used to produce and describe all of our experimental results, which are given in Chapter 9. The design of our software involved the following steps:

1. Identify a set of basic, independent features that most sequential models share.
2. Transform influential techniques from the literature to a unifying control and data structure that decomposes into the basic features.
3. For each basic feature, define the set of interchangeable options corresponding to existing, abstractly distinct implementations of that feature.
4. For each basic feature, try to improve upon the existing solutions, and add those improvements to the set of options.
5. Implement the cross product of the sets of interchangeable options for each feature. Each element in the cross product should be a viable, working on-line modeling algorithm. The cross product will properly include the original algorithms from the literature, and should reproduce their predictions exactly.

Because we combine major components (e.g., state selection and mixtures) and several minor components (e.g., update exclusions and inheritance times) for the first time, and because we added novel options to the component sets (e.g., percolating and bottom-up state selection, and new edge-redirection tree structures), the cross product implements many genuinely novel on-line modeling algorithms. Furthermore, it provides complete experimental control for the evaluation of individual model features.

8.2 The Common Control Structure

The main control loop of the cross-product implementation performs the following computations in sequence for each input symbol, using an adaptive FSM model of the input sequence:

1. Excite the model states whose conditioning context partitions contain the processed input sequence, and make the transition into the next maximum order excited state, initially the root node, and subsequently determined in step 6.
2. Select the coding model, which is represented by one of the excited states.
3. Estimate the probability of the currently scanned symbol using the frequencies at the selected state and possibly its ancestors.
4. Add new descendants to the maximum-order excited state via splitting or redirection of the incoming transition taken, if it is eligible. The maximum-order novel descendant becomes the new maximum order excited state.
5. Add a novel event (i.e., out-transition) corresponding to the currently scanned symbol to each novel excited state or any excited state that "missed." (They will be the highest-order excited states.)
6. Compute the next maximum-order excited state and prepare the next-state transitions into all simulated models. This step also manages the suffix-linked virtual nodes in the string-transition suffix-tree implementation.
7. Update the MDLs at all currently excited states.
8. Compute *before-update* inheritances at excited actual states, if applicable.
9. Update frequencies at excited states.

10. Compute *after-update* inheritances at excited states, if applicable.

8.3 The Cross Product of Distinguishing Features

The algorithmic variants that we test in the Chapter 9 are specified in the column headings of the tables using the conventions described below for describing the major features of modeling algorithms: model structure, probability estimation, frequency updates, and state selection. The description below accomplishes four goals:

- It assigns English names to each feature option, to form a language for precisely describing on-line statistical algorithms.
- The symbols accompanying the English names give a terse labeling system that completely describes on-line algorithms.
- It explains our software's command-line options (see Figure 8.1).
- Lastly, it gives a synopsis of the modeling concepts presented in this work.

8.3.1 Model Structure and Growth

Suffix-tree model structure is determined by the size of the input alphabet, the initial model, any order bounds, and whether the model is implemented with symbol-labelled transitions or string-labelled transitions (which are described in Chapter 3 and denoted by the shorthand '*'). Thus, the following features specify model structure:

Alphabet Bits, b: DMC uses a binary input alphabet, with $b = 1$; the other techniques we test use a 256-ary alphabet, with $b = 8$.

Minimum Order: This global bound o guarantees that at time i, for every unique substring w of $a_1 a_2 \cdots a_{i-1}$ such that $|w| \leq o$, there will be a unique state s in the suffix tree with conditioning context string $\mathbf{context}(s) = w$. Either the suffix-tree has no global order bound (default, given as '-1') or an order bound is specified simply as a scalar (e.g., '3'). The bound is a *minimum* order bound because when a non-zero order bound is combined with one of the edge-redirection criteria other than R_0, portions of the suffix-tree that have already reached the order bound may grow past it.

Order-Zero Initial Model, M: Required to exactly emulate DMC and GDMC. Early predictions are negligibly better with an order -1 initial model, the default.

```
Usage: runDMC [-B(atch)-U(decode)-b-e-s-v-o-r-P-y-z-s-w-i-m-K-x-G-D-M-c] file
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Alphabet_Bits:                      (the log of the input alphabet size)
        -b {1, ..., 16}  (Default = 8)
neg_log_EPSILON: (EPSILON = minimum frequency.  Resolution R of floating point
        arithmetic coder (R >= 32) and Max File Size determine Minimum EPSILON.
        -e {1, ..., R} (DEFAULT = 10 allows a Max File Size of 4.19 M symbols).
                       (Max Frequency = Max File Size = 2^{R-(-log_2(EPSILON))}.
State_Selection:
        -s 0 (Default) select no state
        -s 2 select min-order deterministic excited state; else select no state
        -s 3 select excited state on Min-Entropy frontier; else select no state
        -s 5 same as 3, but approx Min-Entropy frontier top-down;
        -s 6 same as 3, but approx Min-Entropy frontier bottom-up;
State_Selection_Threshold:          (small means nodes mature at young age)
        -v <integer> (Default = 0, actual threshold == n/1024)
Minimum_Order:           (max order of nodes added by string event splitting)
        -o -1 Infinite Order        (PPM_Star)
        -o  0 Zero Order + nodes added by edge reclassification (Default, DMC)
        -o  {1, ..., 32767}         (PPM, PPM-DMC hybrid where higher-than-min-order
                                     nodes are added by eligible event reclassification)
Reclassification_Eligibility;       (for last_taken guest in_event)
        -r 0  Never (Default, PPM, nodes are added ONLY by string-event splitting)
        -r 1  FREQUENCY: if freq(curr_node)+ 1.0 - freq(in_edge) > Thresh'd,
        -r 2  MDL: if guest dest'n->parent->MDL- dest'n->parent->extnsMDL > Thresh_other
        -r 3  Destination Order < Min_Order;
Owner_Edge_Protection:     (don't redirect edges more than once);
        -P  (DEFAULT= off)
Reclassification Thresholds:    (small value speeds growth)
        -y <integer> (edge threshold  = v/1024.0, Default = 0.0)
        -z <integer> (other threshold = w/1024.0, Default = 0.25)
Reclassification_Tree_Structure:   (redirect guest events/edges to new classes/states)
        -t 0  one edge to one new state  (DMC)
        -t 1  chain of linked guest_edges to one new state
        -t 2  chain of linked guest_edges to sibling set of new states
        -t 3  chain of linked guest_edges to suffix chain of new states
Mixture_Weights:        P_est(a|n) = #n/(#n+q) * #a/#n + q/(#n+q) * P_est(a|n->parent)
                                where #n = Sum_{b:count(b|n) >= 1}{#b}, and
        -w A  #a = count(a|n); q = Initial_Mass_Of_Parent_Mixture;
        -w B  #a = count(a|n) - 1; q = |{b: count(b|n) >= 1 }|
        -w C  #a = count(a|n); q = |{b: count(b|n) >= 1 }|
        -w D  #a = count(a|n) - 0.5; q = 0.5* |{b: count(b|n) >= 1 }|
Initial_Mass_Of_Inheritance:        (new nodes'initial escape counts before updates)
        -i <integer> Default: = 0.0(do not use 0.0 with -p A)
Mixture_Inheritance_Time:           (when is P_est(a|n->parent) computed relative to n?)
        -m 0  at model creation Uniform Prior
        -m 1  at  node creation   DMC
        -m 2  at novel event prediction,  Blending:
                                              count(a|n) >=1 => P_est(a|n->parent)=0.0
        -m 3  before novel event updates
        -m 4  after novel event updates         (like 3, slightly more aggressive)
        -m 5  at every event visit
Satisfy_Kirchoff:            Subtract inheritance from parent's distribution.
        -K (DEFAULT = off).         In edge's counts will equal out_edge's counts.
Update_Exclusion:                   (which nodes use in_situ freqs, or subtree freqs too)
        -x 0  Off (Default): every state uses counts gathered from its entire subtree
        -x 1  PPM Update Exclusion: Only selected state uses counts from its subtree.
        -x 2  Maximum-Order Update Exclusion
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Figure 8.1: The command-line options of the executable taxonomy.

Transition Redirection: Two distinct operations are used to build models: transition *splitting* builds string-transition models, while transition *redirection* builds symbol-transition models. Thus the model structure is additionally determined by how and when symbol-labelled transitions are redirected.

Redirection Criteria: These determine the eligibility of the transitions taken into the currently excited states for redirection.

never, R_0 (or equivalently, $*$): Only add nodes by splitting string transitions. (Implements linear-space PPM and PPM*.)

popularity, R_1: Redirect transition if and only if its count exceeds threshold $y/1024$ and the contribution to its destination's count by other in-edges exceeds threshold $z/1024$. (Implements DMC and GDMC.)

MDL, R_2: Redirect transition if and only if the best-performing frontier below its destination's parent has an expected minimal codelength that is $z/1024$ base e bits per character less than that of its parent. (A novel experimental option.)

order, R_3: Redirect transition if and only if the order of its destination is less than the Minimum Order. (Used for full-space implementations of PPM, which, incidentally, are implemented as a variant of DMC.)

Owner Protection, P: this option limits the redirection criterion to "non-owner edges," that is, edges leaving states with Markov order not less than the order of the destination. Prevents transition re-redirection.

Redirection Tree Structure: These options are logically part of the edge redirection criteria, and can redirect the transitions that were taken into the currently excited states as a group. They are easier to understand if they are left whole, rather than decomposed into single-edge redirection criteria. Models built solely by string-transition splitting are not affected by this option.

one-to-one, T_0: If the edge entering the current maximum-order excited state is eligible, redirect it to one new state. (DMC and GDMC.)

many-to-one, T_1: If the edge entering the current maximum-order excited state is eligible, redirect it and all eligible suffix-lined edges above it to the same new state. (A novel, experimental option.)

many-to-children, T_2: If the edge entering the current maximum-order excited state is eligible, redirect it and all eligible suffix-linked edges above it to distinct new states that are children of the original destination state, and siblings of each other. (A novel, experimental option.)

many-to-descendants, T_3: If the edge entering the current maximum order excited state is eligible, redirect it and all eligible suffix-linked edges above it to distinct new states that are suffix-linked descendants of the original destination state. (Emulates PPM and WLZ.)

Redirection Thresholds, y and z: for tuning redirection criteria R_1 and R_2.

8.3.2 Probability Estimation with Mixtures

Chapter 6 explains how all probability estimators in the baseline algorithms can be described as recursive mixtures of the frequency distributions at different excited states. The principal features that distinguish any mixture are when, relative to the state's lifetime, each state's inherited distribution is computed and what weight the inheritance is assigned.

Inheritance Evaluation Time: The *inheritance* at a given state s is a frequency distribution constructed from the frequency data present at its ancestors at a given point in time, relative to s's lifetime.

inherit at model creation, M_0: This is the degenerate mixture that corresponds to every state having a uniform frequency distribution initially.

inherit at state creation, M_1: Required by DMC, but too costly for 256-ary alphabets, since every state will automatically has 256 out-transitions. In contrast, adding the out-transitions as needed results in models where states typically have only 3-5 out-edges.

inherit at novel event prediction, M_2: This is *blending*, the default mixture for PPM variants. Note that this type of "inheritance" differs from the others in that it is forgotten (i.e., set to zero) after its first use.

inherit before novel event updates, M_3: As a short hand, we call this option M when it will not cause confusion.

inherit at every event visit, M_5: Every time a state becomes excited, recompute the frequency distributions conditioned by its ancestors—they

may have changed since last visit.

Initial Mass of Inheritance, i: Let hypothetical variable z be the initial mass of the inheritance, or initial escape count. Then, to correctly emulate PPM variants, z must equal 1.0 for escape method A, and 0.0 otherwise. However, to emulate DMC, each state's initial escape count z must be set to the count on the redirected in-transition. Now, since DMC with lazy cloning can be parameterized to exactly emulate PPM by redirecting out-transitions when they are first created, and therefore have zero counts, the value of z at the destination state can be set to the count on the redirected transition for PPM variants as well, with no loss of correctness. Therefore, to provide lazy DMC variants with the same range of mixture options as PPM variants, we define z to equal the initial mass of the redirected transition, plus i. Note that i must be set to 1.0 to exactly emulate weighting formula A, and 0.0 to exactly emulate B, C, D. But i can be set to other non-negative values as well. We will only specify i when it is not set to the default value that corresponds to the given Mixture Weighting Formula.

Mixture Weighting Formula, A, B, C, or D: These options define mixture weights by determining what gets added to the inheritance mass z (which is initialized as described above) at a given excited state after it "misses," and the initial count k of the added event.

A: z remains unchanged, $k = 0.0$;

B: z is incremented by $1.0, k = -1.0$;

C: z is incremented by $1.0, k = 1.0$;

D: z is incremented by $0.5, k = 0.5$.

Zero-Sum Inheritance, K: If this option is enabled, the inheritance given to a state is subtracted from its parent's distribution. Emulates original DMC. The original binary DMC algorithm did this, which enabled the model transition counts to satisfy Kirchoff's law: the incoming traffic equals the outgoing traffic. But all techniques with mixtures can have this capability.

8.3.3 Frequency Updates

Chapter 5 fully describes three update options, how each of them affects the semantics of the model, and how to emulate them simultaneously in a single model that combines non-trivial mixtures and state selection.

Update Exclusion, X: The default for PPM and PPM* variants is no update exclusion for any states. Otherwise, the short-hand 'X' is included in the column header of PPM variants to denote the selective application of update exclusion to eligible states. When we need to be more specific we use a longer notation:

full updates, X_0: All excited states get a frequency update.

update exclusion, X_1: Only the excited states that failed to recognize the currently scanned input, and the maximum-order state that recognized it, receive frequency updates.

max-order updates, X_2: Only max-order excited state gets a frequency update. Required for exactly emulating DMC and GDMC.

Special GDMC features: These options, shown in Figure 8.2, have no abstract rationalization that we know of, but they are necessary for exactly emulating GDMC, and they can be used with any other algorithm in the cross product.

restriction G: Do not copy predicted out-transition to the out-transition lists of newly added states.

restriction D: Do not update the frequencies at newly added states.

copy depth, C: If any excited states "missed", only the C highest-order excited descendants of the maximum order excited state that "hit" receives a copy of the out-transition corresponding to the currently scanned symbol. To emulate GDMC, let $C = 1$.

8.3.4 The Selection of the Coding Model

The set of state selectors is empirically evaluated in Chapter 9, using a range of state selection thresholds. The percolating state selector mentioned below is presented in Chapter 7. All of our implementations are MDL-based for efficiency, but they could have been implemented using actual entropies of the frequency distributions. We

```
:::::::::::::::::::: Options Included to Emulate GDMC Exactly ::::::::::::::::::::
Dont_Copy_Event_To_Clones:
        -G  (Default = FALSE, copy to clones if ok by Max_Event_Copy_Depth)
Dont_Update_Freqs_At_Clones:
        -D    (Default = False, update all excited nodes, including new ones)
Order_0_Initial_Model:
        -M (Default = FALSE = use order -1 initial model. No -m0 uniform_prior)
            (when combined with -m1, all nodes will have |alphabet| out_events).
Max_Event_Copy_Depth:
        -c 0  copy predicted event all the way to new leaf (Default)
        -c {1,2,..,255} copy event to c min-order extns of predicting_node
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Figure 8.2: The special command-line options added to accommodate GDMC.

have implicitly assumed throughout our work that MDL-based implementations are equivalent in effect to entropy-based implementations.

State Selectors: These options determine how the coding model is dynamically selected for each input symbol.

none, S_0: The default action is to select no state. In this case, the selected order, which we tally for our experiments, is counted as the order of the maximum-order excited state plus one.

heuristic, S_2: Select the min-order excited state with one out-edge [CTW95].

percolating, S_3: See companion paper [Bun97a]. The construction of [WLZ92] (WLZ) uses an order-bounded state selector that produces the same results as our percolating state selector does in a model that has the same structural order bound, assuming that both models use the same selection threshold. As a shorthand, the percolating state selection mechanism is denoted by S when it is the only information-theoretic state selector used in a set of experiments.

top-down, S_5: An MDL-based implementation of the hill-climbing method used in Rissanen's *Context* algorithm.

bottom-up, S_6: A bottom-up complement to S_5.

```
:::::::::::::::::::::::::::::::: Examples::::::::::::::::::::::::::::::::::::
     DMC: -b1 -s0     -r1 -y2048 -z2048 -t0 -o0  -x2 -wA -i0 -m1 -K       -M -c0
    GDMC: -b8 -s0     -r1 -y0    -z256  -t0 -o0  -x2 -wA -i0 -m3    -G -D -M -c1
     PPM: -b8 -s0     -r3               -t3 -o?  -x1 -wC -i0 -m2             -c0
     PPM: -b8 -s0     -r0                   -o?  -x1 -wC -i0 -m2             -c0
    PPM*: -b8 -s2     -r0                   -o-1 -x0 -wC -i0 -m2             -c0
     WLZ: -b8 -s3 -v? -r3                   -o?  -x0 -wA -i1 -m0             -c0
 Context: -b1 -s5 -v? -r0                   -o-1 -x0 -wA -i1 -m0             -c0
Context2: -b8 -s5 -v? -r0                   -o-1 -x0 -wA -i1 -m0             -c0
BestFSMX: -b8 -s3 -v0 -r0                   -o-1 -x1 -wD -i0 -m3             -c0
 BestDMC: -b8 -s0 -v0 -r1 -y1024 -z2048 -t0 -o0  -x1 -wD -i0 -m3          -M -c0
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Figure 8.3: Command lines that execute the Markovian baselines, plus others.

State-Selection Threshold, v: The actual threshold is $v/1024$, and is used to test the difference in expected bits (log base e bits, that is) per character between two models represented by suffix-adjacent states. Higher thresholds cause the algorithm to wait until a state is fairly mature before it is ever used for a prediction.

The correctness of our implementation vis à vis state-selection thresholds warrants comment. The algorithms *Context* and WLZ both call for formulaic state-selection thresholds that are generally a function of the alphabet size (and model order for WLZ), and act as lower bounds to the 'coding penalties' that are theoretically incurred by adding descendants to a state. However, in our preliminary experiments, we found that in order to give these techniques a sporting chance against other techniques, a threshold that never exceeded zero was required. So we replaced the tighter formulaic lower bounds with scalar thresholds, which provide much looser lower bounds, but better results and faster run times.

8.4 The Command-Line

The command-line "usage" message of our program is listed in Figures 8.1 and 8.2 to illustrate how precisely the nomenclature mirrors the actual command-line features. The actual command lines that must be typed to make the cross product emulate the baseline algorithms, ignoring the existence of preprogrammed defaults, are shown

in Figure 8.3. These examples demonstrate the expressive power of our taxonomy—the first six lines plus the final two lines of text completely describe seven different state-of-the-art algorithms[1] and explicitly point out *all* of the abstract differences among them.[2] For each algorithm, the options that are left unspecified do not have any effect on the particular combination of other options.

The remaining two command lines approximate the original binary-alphabet *Context* algorithm [Ris83] and a 256-ary variant [Ris86b]. The emulations are only approximate because the true *Context* model is a non-Markovian FSMX finite state machine, which cannot be represented state-for-state with an FSM that has explicit transitions. However, the true *Context* model is embedded upon the unbounded-order FSMX model constructed by PPM*, since that model contains a state for every substring of the already-processed portion of the input. And since *Context* used top-down hill-climbing state selection, most of the extra states in the Markov FSM will be ignored. If anything, our Markov approximation of *Context* should achieve better performance than the original[3], since the only extra states that will ever be used will be states that tend to improve the performance of the model.

8.5 Summary

Our taxonomical approach to describing and defining novel on-line modeling algorithms constitutes a break with tradition in practical data compression research. Researchers in the past have emphasized (sometimes artificial) differences among their algorithms with memorable acronyms and typically treated existing techniques as "black boxes" (thus often ignoring meaningful parameters) when comparing them to their own constructions. However, quite often the abstract differences among apparently distinct approaches can be expressed in a single technique as different values of parameters that have simple connotations.

---

[1] The seven algorithms are DMC [CH87], GDMC [TR93], PPM [Mof90], PPM* [CTW95], WLZ [WLZ92], plus BestFSMX [Bun97b] and BestDMC, which are the best-performing FSMX and DMC variants tested with this taxonomy so far ([Bun96, Chapter 9]). PPM is listed twice because there are two very different ways to implement it as a suffix tree, both of which produce identical predictions.

[2] Now, *that's* data compression!

[3] comparable performance baselines for Context are unavailable

The early chapters of this thesis transformed the dominant algorithms from the practical literature to the control and data structure that is shared by the dominant algorithms from the information-theoretical literature. Then, the subsequent chapters described in detail the principal groups of components that every online model must have: a means for selecting the coding model, a way to compute probability estimates from the frequency data organized in the model, a scheme for updating that frequency data, and a mechanism and criterion for growing the model structurally. This chapter outlined the subcomponents of each of these parts and combined those components into a single executable cross product. The next chapter uses that cross product implementation to evaluate different solutions to the component subproblems, including the new solutions we introduced earlier.

Chapter 9

PERFORMANCE MEASUREMENTS

In this chapter, we measure the predictive ability of our models by running them on the files of the Calgary Corpus [BCW90]. All probability estimates are coded using a floating-point m-ary arithmetic coder that we based upon the popular integer coder implemented in [WNC87, MSWB93]. Arithmetically coded probability estimates are an excellent measure of model performance because they compute a tight, infinite-precision upper bound on the $-\log$ likelihood of the model, given the input sequence. Furthermore, coding the probability estimates permits verifying that the models are not getting high likelihoods through error or the loss of information by decoding the output and checking that the decoded output is the same as the original input.

All results presented here were produced by the cross-product implementation, and all of the results described here decode correctly. Performance is measured for each file as average output bits per input character ("bpc"). For the Calgary Corpus, input characters are bytes, so the input alphabet size is 256. Performance is summarized for each experiment as average bpc per file, labelled with *Average* or *Avg.* First, we give the known performance of influential algorithms from the literature, and show how our taxonomical reimplementation of those algorithms compares on these baselines. Then in the next two sections, we evaluate the state selection techniques presented in Chapter 7, and the mixtures techniques presented in Chapter 6. Next, we apply the techniques covered in this thesis to improve the compression performance and memory consumption of PPM variants, followed by the performance and memory consumption of DMC variants. Lastly, we discuss issues of universality—whether our conclusions hold in general or just for the test data.

9.1 Baselines

The published performance of on-line stochastic algorithms from the data compression literature that have been implemented are shown in Table 9.1, along with the performance of two popular Unix compression utilities. The utilities are 'compress,'

Table 9.1: The State of the Art in On-Line Statistical Compressors.

File	*Size* (bytes)	LZ78 (compress)	LZ77 (gzip)	DMC	GMDC	PPMC	PPM*
bib	111,261	3.35	2.52	2.28	2.05	2.11	**1.91**
book1	768,771	3.46	3.26	2.51	**2.32**	2.48	2.40
book2	610,856	3.28	2.71	2.25	**2.02**	2.26	**2.02**
geo	102,400	6.08	5.35	**4.77**	5.16	4.78	4.83
news	377,109	3.86	3.07	2.89	2.60	2.65	**2.42**
obj1	21,504	5.23	3.84	4.56	4.40	**3.76**	4.00
obj2	246,814	4.17	2.65	3.06	2.82	2.69	**2.43**
paper1	53,161	3.77	2.80	2.90	2.58	2.48	**2.37**
paper2	82,199	3.51	2.90	2.68	2.45	2.45	**2.36**
pic	513,216	0.97	0.88	0.94	**0.80**	1.09	0.85
progc	39,611	3.87	2.68	2.98	2.67	2.49	**2.40**
progl	71,646	3.03	1.82	2.17	1.83	1.90	**1.67**
progp	49,379	3.11	1.82	2.22	1.90	1.84	**1.62**
trans	93,695	3.27	1.62	2.11	1.73	1.77	**1.45**
Average	224,402	3.64	2.71	2.74	2.52	2.48	**2.34**

Table 9.2: Cross-Product Baselines of Existing Stochastic Techniques.

File	*Size* (bytes)	DMC	GMDC	PPMC (order 3)	PPM*	PPMC (order 5)	PPMD (order 5)
bib	111,261	2.219	2.045	2.114	1.910	1.915	**1.875**
book1	768,771	2.246	2.319	2.478	2.397	2.338	**2.297**
book2	610,856	2.255	2.021	2.271	2.020	2.004	**1.968**
geo	102,400	4.671	5.157	**4.663**	4.828	4.722	4.712
news	377,109	2.894	2.605	2.648	2.419	2.396	**2.364**
obj1	21,504	4.560	4.403	3.766	4.004	**3.736**	3.737
obj2	246,814	3.064	2.817	2.726	2.434	2.446	**2.421**
paper1	53,161	2.889	2.582	2.482	2.373	2.373	**2.336**
paper2	82,199	2.927	2.451	2.457	2.361	2.358	**2.314**
pic	513,216	0.925	**0.803**	0.823	0.854	0.816	0.808
progc	39,611	2.977	2.666	2.499	2.401	2.411	**2.377**
progl	71,646	2.168	1.826	1.904	**1.671**	1.729	1.693
progp	49,379	2.222	1.905	1.843	**1.624**	1.753	1.719
trans	93,695	2.114	1.734	1.772	**1.447**	1.539	1.495
Average	224,402	2.721	2.524	2.460	2.339	2.324	**2.294**

which is based upon Welch's popular implementation [Wel84] of the Ziv and Lempel's second major string-matching algorithm [ZL78], and 'gzip,' which is based upon Ziv and Lempel's first major string-matching construction [ZL77].

Moffat's 1990 implementation, PPMC [Mof90], of Cleary and Witten's 1984 PPM algorithm [CW84b], remained unchallenged until Cleary, *et al.* did away with PPM's order bound to produce PPM* in 1995 [CTW95]. The authors claimed that PPM* outperformed PPMC in their paper. However, PPMC was known to achieve superior compression performance as the order bound increased up to 5 [Mof90], after which its performance starts to decline. In 1993, Howard published a simple change to PPMC's escape mechanism, called PPMD [How93]: add .5 instead of 1.0 to the escape count and scanned event count, whenever a novel event is seen. PPMD gets even better performance than PPMC. Thus, the original PPM* algorithm cannot be called the state of the art until it is shown to perform favorably compared to these higher order PPMC and PPMD parameterizations.

Table 9.2 shows how our emulations of the above statistical techniques perform. The performance is very close to that of the original implementations in Table 9.1. There are slight differences, however, due to our use of floating point frequencies rather than integers in PPM and PPM* and the frequency scaling that is used in PPM, PPM*, DMC, and GDMC, whenever a frequency at a state exceeds some constant maximum value. Our floating-point arithmetic coder enabled us to dispense with frequency scaling. While we were establishing these baselines, we experimented with our implementation of the existing PPMC and PPMD technologies. And, in spite of its longer (unbounded) conditioning contexts, PPM* clearly does not outperform PPM.

9.2 State Selection Experiments

In this section we seek to answer the following questions:

1. Does state selection improve performance in practice?
2. How do the different state selection mechanisms perform relative to each other?
3. What are good threshold values for the information-theoretic techniques?
4. Do the techniques perform predictably for models of different orders?

5. Which improves performance more: state selection, mixtures, or their combination?

The suite of experiments that we ran to answer these questions covered the cross product of the following sets of parameters:

State Selection Techniques in $\{S_0, S_2, S_3, S_5, S_6\}$, where

S_0 is no state selection. The order of the selected state is always the order of the max-order excited state plus one.

S_2 is the approximate state selection used in the original PPM* implementation, where either no state is selected unless the high-order states are *deterministic*, in which case the lowest-order deterministic state is selected. (Deterministic states only recognize one source symbol.)

S_3 is our percolating state selector.

S_5 is top-down hill-climbing state selection.

S_6 is bottom-up hill-climbing state selection.

Probability Estimators in $\{AM_0, DM_3X\}$, where

AM_0 denotes the degenerate mixture that produces an identical uniform prior for each state. In the language of our taxonomy, it combines a constant uniform weighting function on the excited states (mixture-weighting formula A), and evaluates their inherited probabilities *at model creation*, denoted by the inheritance evaluation time M_0.

DM_3X describes one of the better-performing mixtures, combined with update exclusion (X). The mixture-weighting formula (or escape mechanism) is D, while the inheritance evaluation time M_3 equals *inherit before novel event updates.*

FSMX Model Topologies in $\{9^*, 64^*\}$: The model is built using our string- transition suffix-tree construction algorithm (thus the *) to save space and time and to therefore make it possible to evaluate higher-order models[1] at a wide range of state selection thresholds in a reasonable time frame.

[1] Orders 9 and 64 were selected because orders greater than 6 have not been evaluated before and the order 9 experiments of this section can be compared with the results in Section 9.4. Order 9 was selected for those experiments, which among other things, compare model sizes between string-transition and full-space implementations, because it was the largest order that we could evaluate

State Selection Threshold Numerator v, where v is a member of the set $\{-1024, -512, -256, \ldots, 512, 1024\}$. Note that the actual threshold equals $v/1024$, thus the actual thresholds tested performances differences bounded by one bit[2] per character. Recall that the value compared to the threshold during state selection is the difference between a given excited state's expected codelength per source symbol, and the expected codelength per symbol at a complete frontier below the state. In hill-climbing techniques, the frontier is formed by the children of the state, while the percolating technique uses the complete frontier with the minimal codelength.

All other features of the models were held constant throughout the experiment. The results of the experiment are summarized in Tables 9.3 and 9.4. Table 9.3 compares performance of state selection techniques on a vanilla FSM model with Markov order 64 on the files of the Calgary Corpus. Table 9.4 compares performance of state selection techniques on an otherwise identical model that also performs one of the better-performing mixtures. At the bottom of each table we summarize the performance on the Corpus for the order 64 models, denoted by the column headers, and for otherwise identical models with Markov order 9. Next to the bits per character figure for each file and model, we give the average selected order of that model. The average order of the maximum order excited state in all models can be computed from the average selected order of the corresponding model that uses no state selection (S_0), by subtracting one from that value.

Now we are prepared to answer the questions posed at the beginning of this section:

1. All forms of state selection tested improve performance. The higher the Markov order, the greater the improvement. The improvement due to state selection is greater if the model does not use one of the better-performing mixtures.

---

using the full-space PPM implementation. Order 64 was selected because it is large enough to allow extrapolation to unbounded order models (some of which are evaluated in Section 9.4), but does not bog down on the extremely high order models constructed for the file "pic" when there is not an order bound.

[2] Recall that a state's performance is recorded as sums of the $-\log$ of the probability estimates. Because the the existing floating point library provides only provide a base e logarithm function, and converting base e logarithms to base 2 is an unnecessary multiplication, we do not do so for the MDLs stored at states.

Table 9.3: Effect of different state selection techniques on the compression performance and average selected order of a vanilla order-64 FSMX model without blending or Mixtures.

File	$A^*64M_0S_0$		$A^*64M_0S_2$		$A^*64M_0S_6v_0$		$A^*64M_0S_5v_{-16}$		$A^*64M_0S_3v_0$	
		Select		*Select*		*Select*		*Select*		*Select*
	(bpc)	*Order*	(bpc)	*Order*	(bpc)	*Order*	(bpc)	*Order*	(bpc)	*Order*
bib	2.749	12.63	2.558	5.35	2.435	3.99	2.312	3.41	**2.286**	3.32
book1	3.452	8.32	3.374	6.49	2.873	4.28	2.444	3.13	**2.432**	2.85
book2	2.857	10.46	2.715	6.45	2.461	4.63	2.198	3.59	**2.189**	3.44
geo	6.059	4.54	6.040	3.42	5.548	2.12	5.280	2.20	**5.216**	2.01
news	3.302	13.12	3.170	5.88	2.999	4.22	2.822	3.50	**2.794**	3.30
obj1	4.584	12.98	4.526	7.87	4.579	3.18	**4.507**	7.55	4.531	3.37
obj2	2.989	14.33	2.851	5.19	2.885	3.89	2.845	3.79	**2.829**	3.87
paper1	3.208	8.98	3.097	4.80	2.944	3.54	2.756	2.77	**2.748**	2.77
paper2	3.300	8.00	3.204	5.23	2.911	3.62	**2.588**	2.67	2.590	2.59
pic	1.148	47.13	1.052	38.34	0.898	19.67	0.993	35.37	**0.852**	5.19
progc	3.143	9.12	3.013	4.56	3.002	3.40	2.888	3.02	**2.883**	2.98
progl	2.314	19.26	2.098	6.86	2.094	4.38	2.066	4.73	**2.044**	3.85
progp	2.224	20.92	1.975	5.63	2.002	4.61	1.989	3.54	**1.976**	4.19
trans	2.025	25.66	1.775	5.71	1.776	4.29	1.756	3.90	**1.727**	3.95
Average	3.097		2.961		2.815		2.675		**2.650**	
Average for order-9 models:										
	3.020		2.946		2.804		2.674		**2.654**	

Table 9.4: Effect of different state selection techniques, on the compression performance and average selected order of an order-64 FSMX model with Update Exclusion (X) and Mixtures (M_3, D).

File	D*64M_3S_0		D*64M_3S_2		D*64$M_3S_6v_0$		D*64$M_3S_5v_{-8}$		D*64$M_3S_3v_0$	
		Select		*Select*		*Select*		*Select*		*Select*
	(bpc)	*Order*	(bpc)	*Order*	(bpc)	*Order*	(bpc)	*Order*		*Order*
bib	2.019	12.63	1.828	5.35	1.816	5.10	1.801	4.19	**1.788**	4.54
book1	2.428	8.32	2.407	6.49	2.332	5.74	**2.205**	4.38	**2.205**	4.44
book2	2.099	10.46	1.970	6.45	1.937	5.95	1.878	4.72	**1.876**	5.09
geo	4.727	4.54	4.756	3.42	4.705	2.97	4.550	2.79	**4.508**	2.28
news	2.452	13.12	2.347	5.88	2.329	5.37	2.301	4.57	**2.292**	4.62
obj1	3.793	12.98	3.766	7.87	3.757	7.39	3.720	7.53	**3.699**	3.29
obj2	2.422	14.33	2.280	5.19	2.288	4.99	2.287	4.59	**2.272**	4.74
paper1	2.411	8.98	2.302	4.80	2.288	4.46	2.256	3.67	**2.249**	3.96
paper2	2.394	8.00	2.326	5.23	2.288	4.75	**2.218**	3.65	2.221	3.92
pic	0.969	47.13	0.849	38.34	0.785	35.62	0.819	35.74	**0.795**	21.13
progc	2.440	9.14	2.304	4.56	2.310	4.28	2.307	3.88	**2.296**	4.08
progl	1.766	19.26	1.530	6.86	1.539	5.67	1.554	5.92	**1.528**	5.30
progp	1.767	20.92	1.504	5.63	1.516	5.46	1.550	4.85	**1.505**	5.39
trans	1.611	25.66	1.306	5.71	1.306	5.50	1.315	4.68	**1.291**	4.88
Average	2.378		2.248		2.228		2.197		**2.180**	
Average for order-9 models:										
	2.281		2.250		2.235		2.202		**2.191**	

(Mixtures hedge against local order over-estimation by including data from lower order states in the estimate.)

2. The performance increases of each state selector, relative to the vanilla order-64 model, were: 4.4% for S_2, 9.1% for S_6, 13.6% for S_5, and 14.4% for S_3. The selectors obtained about 2% less of a performance increase in similar order-9 models.

   The performance increases of each state selector, relative to the order-64 model with mixtures but no state selection, were: 5.5% for S_2, 6.3% for S_6, 7.6% for S_5, and 8.3% for S_3. The selectors obtained about 4% less of a performance increase in similar order-9 models.

3. The percolating and bottom-up selectors always perform best with their respective state selection thresholds set to zero. The top-down hill-climbing selector performs best when its threshold is set to a small negative value. This compensates for its tendency to underestimate local order. The ideal value is a function of the true order of the input sequence. For most files of the Calgary Corpus the top-down selector did best with a values between $-8/1024$ and $-32/1024$. However, with this selector a threshold that does best for the low-order files would get worse performance on the higher-order files, and *vice versa*. Since the corpus is dominated by higher-order text files, these negative values were shown to be the best by our experiment.

4. The performance ranking of the selectors is consistent for all Markov orders that we have tested, regardless of whether the models use mixtures or not. The percolating state selector S_3 consistently outperforms the other selectors on all files and at all Markov orders that we have tested. However, the top-down hill-climbing selector S_5 can be parameterized to perform nearly as well.

5. In spite of the consistent behaviors of the state selectors, *mixtures* (including blending), provide about one and a half times the performance increase to a vanilla model than does the best state selector. The mixture used in this test improved the performance of the order-64 vanilla model by 23.2% and the order-9 vanilla model by 24.5%.

In summary, a good mixture works better than the best state selector, but the combination is better still. The best-performing state selector is our percolating tech-

nique, followed closely by top-down hill climbing—but only if it is correctly parameterized. An unbounded-order model combined with the percolating state-election technique satisfies a primary goal of universal on-line modeling: doing away with model parameters that the modeling algorithm cannot automatically deduce from the input sequence.

9.3 Mixture Experiments

In this section we address the following questions:

1. Which mixtures perform best?
2. How do the various mixture weighting formulae and inheritance times interact?
3. Is the effectiveness of update exclusion affected by the mixture with which it is combined?

Table 9.5 shows the relative effectiveness of most combinations of mixture weighting functions and inheritance evaluation times. Inheritance time M_1 (*inherit at state creation*) was omitted because it exhibits impractical space consumption for models with 256-character input alphabets. As expected from past experience with PPM, weight functions C and D produce the best results.

Table 9.6 is a study on the value of using update exclusion, especially in models using state selection. This study is important because the largest cost of correctly implementing state selection with lazily evaluated model refinements, or of combining approximate state selection with mixtures, is keeping two counts for every transition: an update-excluded count and a full-update count. For example, since the percolating state selector S_3 improves an order-9 model with mixtures DM_3i_0 and update exclusion X_1 by about 8.3%, and update exclusion improves an order-9 model with mixtures DM_3i_0 and percolating state selector S_3 by about 6.5%, it probably would not be worth the trouble to implement state selection if doing so would require disposing of update exclusion. Basically, the later the inheritance evaluation time, the more update exclusions improve performance.

Table 9.7 shows how the better performing mixtures, highlighted in **bold** in Figure 9.5, perform on individual files.

Our controlled component-wise experiments with the Calgary Corpus show that the best-performing mixtures outperform models that assume a uniform prior fre-

Table 9.5: How average compression performance on the Calgary Corpus as a whole is affected by varying mixture inheritance times and mixture weight functions, in models with and without (percolating) state selection.

Inherit Time	*A*9X*	*B*9X*	*C*9X*	*D*9X*	*A*9XS*	*B*9XS*	*C*9XS*	*D*9XS*
M_0	2.965	4.695	2.825	2.767	2.602	2.925	2.522	2.508
M_2	**2.631**	**2.505**	2.365	2.306	**2.386**	2.674	2.227	2.206
M_3	2.767	2.688	2.329	**2.281**	2.475	2.559	**2.197**	**2.191**
M_5	2.851	2.520	**2.300**	2.302	2.482	**2.419**	2.203	2.238

Table 9.6: The percent improvement of models using update exclusion over the same model variants without update exclusion.

Inherit Time	*A*9X*	*B*9X*	*C*9X*	*D*9X*	*A*9XS*	*B*9XS*	*C*9XS*	*D*9XS*
M_0	1.8	-6.5	1.1	2.9	2.0	0.2	6.9	2.1
M_2	5.5	3.2	2.2	5.9	5.7	3.7	3.2	5.4
M_3	5.9	-0.8	2.9	7.5	5.5	2.7	3.8	6.5
M_5	5.8	5.3	4.3	10.2	1.4	6.2	4.3	6.4

Table 9.7: Compression performance for the best inheritance times given each weighting mechanism

	size	*A*9X*	*B*9X*	*C*9X*	*D*9X*	*A*9XS*	*B*9XS*	*C*9XS*	*D*9XS*
File	*(bytes)*	M_2	M_2	M_5	M_3	M_2	M_5	M_3	M_3
bib	111,261	2.089	2.129	1.895	1.884	1.910	2.025	1.809	**1.794**
book1	768,771	2.576	2.438	2.391	2.393	2.223	2.238	**2.194**	2.198
book2	610,856	2.175	2.152	1.987	1.996	1.938	1.990	1.876	**1.871**
geo	102,400	6.081	4.460	4.842	4.724	4.939	**4.432**	4.455	4.511
news	377,109	2.666	2.623	2.385	2.363	2.465	2.546	2.300	**2.298**
obj1	21,504	4.555	3.973	3.785	3.737	4.343	3.956	**3.654**	3.704
obj2	246,814	2.755	2.699	2.368	2.340	2.657	2.657	2.317	**2.302**
paper1	53,161	2.579	2.652	2.347	2.340	2.397	2.579	2.268	**2.250**
paper2	82,199	2.552	2.527	2.354	2.347	2.299	2.412	2.232	**2.219**
pic	513,216	0.889	0.804	0.817	0.807	0.820	**0.781**	0.790	0.797
progc	39,611	2.669	2.728	2.373	2.365	2.514	2.650	2.305	**2.301**
progl	71,646	1.777	1.962	1.600	1.590	1.677	1.878	1.576	**1.548**
progp	49,379	1.878	2.055	1.652	1.659	1.737	1.940	1.603	**1.566**
trans	93,695	1.598	1.864	1.396	1.389	1.483	1.786	1.377	**1.320**
Avg.		2.631	2.505	2.300	2.281	2.386	2.419	2.197	**2.191**

quency distribution at every state by about 23% in models without state selection, and by about 16% in models that use state selection, on the Calgary Corpus. We also demonstrated how mixtures interact with update exclusion, which improves performance as much as $6-10\%$. Broadly speaking, the later the inheritance evaluation time, the greater the impact of update exclusion.

Not surprisingly, the overall best-performing weighting formulas (escape mechanisms) were C and D. Regardless of the other parameters, mixtures that *inherit before novel event updates* consistently outperform blending by about 1%, when used with the competitively performing mixture weights C and D or with state selection.

9.4 Improvements to PPM Variants

In this section, we demonstrate that our transition-splitting suffix-tree implementation reduces space requirements for PPM models, and our enhancements consistently and cumulatively improve the probability estimates of PPM and PPM* models, well beyond the current state of the art.

Given a shorthand version of our nomenclature, the standard reference version of PPM [Mof90] becomes PPM*C3X* if it is implemented with symbol transitions, and PPM*C*3X* if implemented with string transitions. The prior state-of-the-art PPM variant, which has order 5, update exclusion, and uses escape mechanism D, is PPM*D5X* or PPM*D*5X,* depending on the suffix-tree implementation. Lastly, the original PPM* algorithm is distinguished from other variants by describing it as PPM*C**.

Table 9.8 shows the effects of progressively applying the optimizations covered in this thesis (update exclusion X, percolating state selection S, and mixtures that *inherit before updates* M) to PPMC and PPMD with order 9. Table 9.9 shows the effect of progressively applying the same optimizations to PPM*. Order 9 (the maximum order symbol-transition structure that our machines could handle) was selected for these experiments to demonstrate the fact that with our optimizations, increased model order corresponds to increased performance, even past order 5. Observe that the improvements by the three optimizations are consistent, significant, and cumulative, for all variants and all files.

Tables 9.8 and 9.9 are divided into right and left halves, with the left halves showing the results of using mixture weighting function C, and the right halves showing

Table 9.8: Compression performance for PPM variants given as bits per character (bpc).

File	*C9*	*C9X*	*C9XM*	*C9XS*	*C9XSM*	*D9*	*D9X*	*D9XM*	*D9XS*	*D9XSM*
bib	2.024	1.984	1.951	1.836	1.809	2.045	1.940	1.915	1.811	1.792
book1	2.444	2.504	2.421	2.234	2.193	2.444	2.455	2.388	2.210	**2.180**
book2	2.099	2.106	2.047	1.918	1.878	2.107	2.058	2.011	1.894	1.864
geo	4.845	4.719	4.709	4.493	4.456	4.868	4.709	4.724	4.474	4.461
news	2.491	2.453	2.412	2.337	2.302	2.533	2.417	2.389	2.314	2.292
obj1	4.035	3.757	3.726	3.689	3.657	4.146	3.757	3.745	3.695	3.681
obj2	2.534	2.424	2.402	2.344	2.324	2.589	2.392	2.380	2.325	2.314
paper1	2.453	2.432	2.391	2.311	2.275	2.487	2.395	2.364	2.287	2.261
paper2	2.434	2.448	2.391	2.276	2.236	2.452	2.404	2.359	2.249	2.217
pic	0.813	0.828	0.818	0.797	0.789	0.819	0.814	0.808	0.791	0.786
progc	2.502	2.452	2.418	2.346	2.314	2.551	2.420	2.395	2.325	2.304
progl	1.787	1.710	1.673	1.609	1.578	1.822	1.667	1.637	1.584	1.559
progp	1.810	1.773	1.747	1.640	1.617	1.830	1.731	1.710	1.616	1.599
trans	1.573	1.520	1.500	1.399	1.382	1.582	1.468	1.454	1.365	1.353
Avg.	2.417	2.365	2.329	2.231	2.201	2.448	2.330	2.306	2.210	2.190

Table 9.9: Compression performance for PPM* variants given as bits per character (bpc).

File	*C**	*C*X*	*C*XM*	*C*XS*	*C*XSM*	*D**	*D*X*	*D*XM*	*D*XS*	*D*XSM*
bib	1.910	1.864	1.830	1.825	1.798	1.945	1.836	1.812	1.803	**1.786**
book1	2.397	2.453	2.369	2.237	2.195	2.406	2.415	2.346	2.214	2.184
book2	2.020	2.023	1.963	1.915	1.875	2.040	1.989	1.940	1.894	**1.862**
geo	4.828	4.702	4.692	4.489	**4.453**	4.850	4.689	4.705	4.469	4.458
news	2.419	2.383	2.342	2.328	2.293	2.465	2.355	2.327	2.308	**2.285**
obj1	4.004	3.837	3.807	3.683	**3.651**	4.115	3.812	3.799	3.692	3.678
obj2	2.434	2.323	2.303	2.313	2.293	2.494	2.299	2.288	2.294	**2.283**
paper1	2.373	2.344	2.302	2.302	2.266	2.409	2.314	2.283	2.279	**2.250**
paper2	2.361	2.369	2.312	2.272	2.231	2.383	2.333	2.289	2.244	**2.213**
pic	0.854	1.066	1.049	0.795	0.786	0.842	0.983	0.971	0.789	**0.781**
progc	2.401	2.341	2.308	2.333	2.301	2.454	2.318	2.294	2.314	**2.291**
progl	1.671	1.600	1.564	1.594	1.561	1.714	1.572	1.542	1.572	**1.545**
progp	1.624	1.572	1.548	1.575	1.552	1.657	1.543	1.524	1.550	**1.531**
trans	1.447	1.391	1.372	1.369	1.352	1.464	1.353	1.337	1.337	**1.325**
Avg.	2.339	2.305	2.269	2.216	2.186	2.374	2.272	2.247	2.197	**2.177**

Table 9.10: Model Size and Topology for PPM* variants.

File	*Size* (Bytes)	*Nodes* PPM*	*Edges* PPM*	*Avg. Order* PPM*	*Average Select Order* *C*,D**	*C*XS*	*C*XSM*	*D*XS*	*D*XSM*
bib	111,261	59,845	171,361	11.85	5.35	4.32	4.34	4.36	4.37
book1	768,771	385,283	1,154,307	7.32	6.49	3.93	4.14	4.07	4.26
book2	610,856	324,528	935,639	9.60	6.46	4.64	4.77	4.77	4.89
geo	102,400	27,712	130,364	3.54	3.42	1.92	1.92	1.97	1.95
news	377,109	196,337	573,701	18.15	5.91	4.25	4.32	4.36	4.40
obj1	21,504	7,025	27,797	54.22	29.21	3.02	3.01	3.05	3.05
obj2	246,814	133,360	380,421	18.03	5.22	4.50	4.50	4.53	4.55
paper1	53,161	29,040	82,456	8.04	4.80	3.71	3.75	3.77	3.79
paper2	82,199	43,213	125,667	7.03	5.24	3.50	3.69	3.58	3.74
pic	513,216	274,879	752,034	2353.38	1848.72	9.59	11.35	19.73	21.25
progc	39,611	21,174	61,040	8.27	4.56	3.79	3.80	3.82	3.84
progl	71,646	46,507	118,408	24.65	6.92	4.95	5.04	5.08	5.15
progp	49,379	33,068	82,702	58.75	5.65	5.17	5.19	5.22	5.23
trans	93,695	66,605	160,339	57.34	6.01	4.60	4.63	4.71	4.75
Avg.	224,402	117,755	339,731	188.08	138.85	4.42	4.60	5.21	5.37

results for D. Note that while escape mechanism D was known to improve PPM's performance, it was believed to not improve PPM*. Indeed, column D* shows that D hurts the performance of PPM* (shown in column C*). However, the respective differences between the columns labeled *C9, C*, D9, D** and *C9X, C*X, D9X, D*X* clearly show that the factor that actually determines the applicability of D over C is the use of update exclusion, not the presence of an order bound. On the other hand, the combination of optimizations tested here significantly reduce the performance difference that D and C impose in PPM and PPM* variants. That is, the greater than 1% performance improvement of D over C in simpler variants is more than halved in *XSM* variants. Since D, C, and other known solutions to the "zero frequency problem" are known to have no principled basis [WB91], our optimizations increase the universality of PPM variants by reducing the relative effects of the necessarily *ad hoc* solutions to the zero frequency (i.e., mixture weighting) problem.

Table 9.10 summarizes the structure of the unbounded-order models constructed by PPM*, and their average selected local orders, as determined by both the state selection mechanism presented in this paper and the original mechanism used with

Table 9.11: Model Size, Topology, and Performance of Order-Bounded PPM* vs. PPM

File	*Nodes*		*Edges*		*Avg Order* PPM*D*	*Avg Select Order* PPM*DXSM*		*Avg bpc* PPM*DXSM*	
	PPM9	PPM*9	PPM9	PPM*9	9,*9	9	*9	9	*9
bib	256,060	28,780	322,124	94,844	6.70	5.03	4.16	1.792	1.794
book1	1,751,574	273,100	2,364,611	886,137	6.72	4.39	4.30	2.180	2.198
book2	1,201,344	174,208	1,596,191	569,055	7.03	5.13	4.74	1.864	1.871
geo	606,519	27,360	706,155	126,996	3.08	2.14	2.23	4.461	4.451
news	1,034,701	112,111	1,296,732	374,142	6.26	5.07	4.35	2.292	2.298
obj1	98,879	4,899	115,263	21,283	4.40	3.96	3.29	3.681	3.704
obj2	688,520	61,979	840,780	214,239	6.21	5.43	4.30	2.314	2.302
paper1	171,916	19,250	213,383	60,717	5.77	4.51	3.79	2.261	2.250
paper2	250,416	30,920	317,083	97,587	5.95	4.23	3.79	2.217	2.219
pic	372,069	28,325	449,835	106,091	8.28	6.95	6.66	0.786	0.797
progc	131,007	13,448	160,837	43,278	5.69	4.69	3.87	2.304	2.300
progl	139,043	17,003	174,279	52,239	7.06	5.81	4.65	1.559	1.548
progp	101,018	11,378	125,325	35,685	6.95	5.60	4.61	1.599	1.566
trans	147,943	16,322	183,008	51,387	7.42	5.76	4.17	1.353	1.320
Avg.	496,500	58,506	633,258	195,263	6.25	4.91	4.24	2.190	2.191

PPM*. The *Nodes* column refers to the number of actual states, while the *Edges* column refers to the number of transitions. The *Avg Order* column gives the average order of the maximum-order excited state for each input symbol; similarly for *Avg Selected Order*. Note that the compression improvement to PPM* variants by our state-selection mechanism is less than for PPM variants (roughly 3.5% vs 5.5%), but then observe in columns *C** and *D** that PPM*'s default state-selection mechanism does reduce PPM*'s average order significantly. Recall from Section 5.1 that the higher-order virtual nodes do not reduce the model's expected codelength. Thus for PPM* with its original heuristic state selector, less potential improvement is available to optimization *S*.

Finally, Table 9.11 shows the tradeoffs of using a string-transition implementation over a symbol-transition implementation. First, there is no cost, other than increased design complexity, of using string transitions if information-theoretic state selection is not used. In that case, the probability estimates, visited orders, and selected orders are identical to those of a symbol-transition implementation. However, with information-theoretic state-selection, a very small difference in the performance (0.1%) of the two implementations occurs. This difference is due solely to local order-underestimation that is brought about by the approximate deduction of codelengths at newly split virtual nodes described in Section 7.6.1. A more careful approximation than ours is possible and can tighten the order estimates. For example, it is easy to deduce whether an excited virtual node was added after its virtual ancestors and whether an excited virtual node has more than one virtual child.

The benefit of the string-transition implementation is a significant space savings, which depends upon the input file, order bound, and the implementation, but which generally amounts to at least half the cost of a symbol-transition implementation. The true space savings (in bytes) provided by the string-transition implementation, relative to an equivalent symbol-transition implementation, equals

$$(V - V^*) \cdot \text{size}(node) + E \cdot \text{size}(edge) - (E^* \cdot \text{size}(edge^*) + F),$$

where V (V^*) is the number of states in the symbol-transition (string-transition) model, E (E^*) is the number of *edges* (*edge*s*), and F equals the input file size plus a small amount of buffering overhead. The sizes of model states (*nodes*), and symbol transitions (*edges*) or string transitions (*edge*s*), are implementation-dependent.

9.5 Improvements to DMC Variants

9.5.1 GDMC and LazyDMC

We began our attempts to create a better-performing variant of DMC [CH87] with two starting points: the GDMC algorithm [TR93] and our own lazy-cloning DMC variant, LazyDMC, proposed in [Bun94].

GDMC uses a mixture with weight function A (thus nothing gets added to a state's escape count after it is initially set to the redirected prefix edge's count) and inheritance evaluation time M_3. In addition, GDMC uses the three restrictions on the frequency updates and number of event copies made, G, D, and C_1, which are described in Section 8.3.3. A surprisingly important difference between GDMC, DMC, and LazyDMC is that GDMC essentially eliminates one of the cloning thresholds: a transition is eligible for redirection if and only if the counts contributed to its destination by other entering edges exceed $z/1024$. In preliminary experiments, we found that in GDMC with the single transition redirection threshold but with each combination of options G,D, and c_1 removed, performance dropped drastically, to an unacceptable 3.5 bpc. Furthermore, the Markov order of the model dropped very close to 1, on average.

The taxonomical shorthand for GDMC's structure is $b_8 S_0 R_0 T_0 y_0 z_{256} GDMC_1$; and for its mixtures and updates $AM_3 X_2 i_0$. We will use the term "GDMC" to denote any algorithm that is based upon the above options. In a GDMC variant, additional symbols will be added to the above string to specify the addition or replacement of command line options. For example, "GDMC $PS_2 z_{1024}$" is GDMC with Owner Protection, heuristic state selection, and a redirection threshold of 1024/1024.

GDMC calls for maximum-order updates, X_2, but our experiments found that the difference in performance between X_2 and X_1 is negligible. GDMC uses slightly fewer nodes and edges with X_1.

The key differences between GDMC and LazyDMC are given respectively by $y_0 AGDC_1$ and $y_{1024} D$. LazyDMC uses a mixture with weight function D[3] and inheritance evaluation time M_3. The only difference between binary DMC and LazyDMC,

[3] Recall that DMC variants set the escape counts at a novel state to the count of the redirected prefix edge, initially, and that option D will cause .5 to be added to the escape count every time a novel event is seen.

Table 9.12: Dueling 256-ary DMC baselines: GDMC and LazyDMC.

	GDMC				LazyDMC			
File	nodes	edges	bpc	select order	nodes	edges	bpc	select order
bib	81,021	79,539	**2.045**	6.67	26,346	58,392	2.113	5.89
book1	591,590	581,455	**2.319**	6.82	163,770	445,133	2.397	5.30
book2	452,474	442,377	**2.021**	7.25	133,512	328,945	2.118	5.98
geo	69,882	77,511	5.157	3.72	17,616	81,220	**4.602**	3.30
news	284,713	281,292	**2.605**	6.84	84,992	231,301	2.610	5.93
obj1	15,708	14,957	4.403	39.35	4,035	16,294	**3.868**	12.15
obj2	167,523	179,104	2.817	6.00	56,606	146,354	**2.654**	5.66
paper1	40,087	39,341	2.582	5.37	12,955	31,207	**2.568**	4.61
paper2	62,235	60,996	**2.451**	5.72	19,049	48,482	2.492	4.66
pic	393,753	364,703	**0.803**	1825.45	66,391	152,121	0.831	185.05
progc	29,428	29,155	2.666	5.25	9,554	23,376	**2.616**	4.55
progl	53,299	52,550	**1.826**	7.94	18,066	36,433	1.892	7.18
progp	35,077	34,969	1.905	6.73	11,956	24,019	**1.891**	6.01
trans	67,167	67,278	**1.734**	7.75	24,145	43,938	1.801	6.95
Avg.	162,216	164,659	2.524		46,357	119,087	**2.461**	

other than the alphabet-size, is that LazyDMC lazily evaluates the out-transitions that are copied to a newly cloned state from its parent. LazyDMC does not add any restrictions, in contrast to GDMC, which merges the cloning thresholds into one threshold and then restricts the edge-copying and frequency update mechanism. The cloning thresholds that we found to work well with LazyDMC were $y/1024 = 1$ and $z/1024 = 2$.

The taxonomical shorthand for LazyDMC is $b_8 MS_0 R_0 T_0 y_{1024} z_{2048}\ DM_3 X_1 i_0$. We shall use "LazyDMC" to denote the above, appended with the appropriate symbols to denote any variations as we did for "GDMC."

The relative performance, model size, and average model order of GDMC and LazyDMC are shown in Table 9.12. LazyDMC performs about 2.6% better than GDMC, and constructs a model that is about 50% smaller. Recall that the average selected order of both baselines is equal to the average order of the maximum-order excited state plus one.

Table 9.13: The effect of *owner protection* (P) on GDMC and LazyDMC

	GDMC P				LazyDMC P			
File	nodes	edges	bpc	select order	nodes	edges	bpc	select order
bib	53,023	56,057	**2.022**	6.80	21,509	43,276	2.069	6.34
book1	404,359	420,588	2.352	6.96	134,026	306,794	**2.336**	5.97
book2	286,264	302,578	**2.019**	7.43	104,622	220,773	2.063	6.62
geo	57,890	59,435	5.383	3.70	15,349	60,880	**4.594**	3.40
news	191,306	199,725	2.584	6.98	67,780	155,706	**2.560**	6.35
obj1	11,638	11,872	4.401	39.35	3,357	12,707	**3.862**	12.7
obj2	112,897	121,202	2.733	6.35	43,998	98,042	**2.592**	6.09
paper1	28,687	30,142	2.570	5.45	10,802	23,849	**2.548**	4.78
paper2	43,912	46,233	**2.443**	5.82	15,914	35,778	2.452	5.02
pic	106,517	109,608	0.932	1825.83	30,628	73,225	**0.778**	188.18
progc	21,104	22,116	2.644	5.40	8,092	18,172	**2.594**	4.76
progl	33,274	35,444	**1.808**	8.16	14,148	26,043	1.863	7.76
progp	22,689	24,273	**1.875**	6.96	9,700	18,081	1.880	6.42
trans	43,650	46,844	**1.680**	8.02	19,479	32,874	1.767	7.40
Avg.	101,229	106,151	2.532		35,672	80,443	**2.426**	

9.5.2 Owner-Protected DMCs

The first optimization we tried was to prevent edges from being redirected more than once with the *owner protection* option, P. The results are shown in Table 9.13. Owner protection very slightly lowered GDMC's performance, but it improved LazyDMC's performance by 1.4%. Most significantly, the option reduced model sizes by roughly 35% and 25% for each technique respectively.

9.5.3 Owner-Protected DMCs with State Selection

We applied the second optimization, state selection, to the owner-protected variants of the originals. Tables 9.14 and 9.15 show the results of applying state-selection to owner-protected GDMC and LazyDMC. Percolating state selection improved owner-protected LazyDMC's performance by 1.7%, while the other state selectors gave about 1.3% improvement. Percolating state selection improved owner-protected GDMC's performance by 1.0%.

We tested the addition of all state-selectors to owner-protected GDMC using

both types of update exclusion: maximum-order updates X_2 and an regular update exclusion X_1. For all state selectors, we used the selection threshold $v = 0$. We give results for X_1 so that comparison with LazyDMC is focused upon how in the two technique's particular combinations of mixtures and edge-redirection criteria are affected by state selection. The average bits per character for owner-protected GDMC with X_2 were 2.540, 2.498, 2.549, and 2.505 for the state selectors s_2, s_3, s_5, and s_6 respectively. These figures indicate that for lazily evaluated DMC variants, the choice between regular update exclusion and max-order updates makes little difference upon compression performance.

The experiments presented here constitute a good start at improving DMC's performance. LazyDMC has compression performance superior to GDMC's, and our experiments indicate that eliminating the minimum transition-count cloning threshold and restricting the production of edge copies as a counter-measure is a flawed approach. LazyDMC, with owner protection and percolating state selection, builds a model that is half the size of GDMC's model and which gets 5.6% better performance than GDMC and 12% better than DMC. However, the small gains from adding state selection to DMC models are probably not worth the cost. Furthermore, these resulting models are still not competitive with our improved PPM and PPM* variants.

There are several promising suites of experiments left to try on LazyDMC:

1. MDL-based edge redirection instead of popularity-based edge redirection, with and without percolating state selection;

2. the many-to-one and many-to-siblings transition redirection tree structures, with and without percolating state selection; and

3. the above alternative tree structures, with and without percolating state selection, with MDL-based edge redirection.

The above list is actually a fairly large body of experiments, since edge-redirection and state-selection thresholds will have to be varied within each suite.

Lastly, one problem that definitely warrants solution is the excessive run time of state-selection techniques and mixtures on the file 'pic,' which induces a GDMC model with very high Markov order. A simple solution is to add another parameter to the taxonomy, Maximum Order, and set it to a large number like, say 64, which will not have a significant affect on compression performance, but will greatly reduce run time.

Table 9.14: LazyDMC with owner protection and state selection.

	PS_2		PS_3		PS_5		PS_6	
File	bpc	select order	bpc	select order	bpc	select order	bpc	select order
bib	2.020	4.88	**2.018**	4.14	2.031	3.95	2.019	4.34
book1	2.332	5.59	**2.298**	4.35	2.300	4.14	2.319	4.83
book2	2.037	5.74	**2.030**	4.80	2.035	4.57	2.035	5.08
geo	4.599	3.12	**4.484**	2.27	4.488	1.97	4.600	2.76
news	**2.533**	5.31	2.534	4.25	2.540	4.07	**2.533**	4.53
obj1	3.848	10.05	**3.830**	2.82	3.851	2.58	3.849	10.80
obj2	**2.548**	4.87	2.560	4.31	2.567	4.17	2.551	4.41
paper1	**2.520**	4.20	2.525	3.57	2.532	3.37	2.525	3.71
paper2	2.435	4.53	**2.429**	3.60	2.435	3.42	2.436	3.96
pic	0.761	126.25	0.758	41.57	0.779	7.75	**0.752**	181.25
progc	**2.557**	4.01	2.575	3.63	2.581	3.50	2.565	3.63
progl	**1.805**	6.32	1.822	4.57	1.833	4.35	1.807	4.80
progp	**1.821**	5.13	1.840	4.60	1.853	4.38	1.828	4.63
trans	**1.690**	5.24	1.708	4.40	1.720	4.13	1.692	4.73
Avg.	2.393		**2.386**		2.396		2.394	

9.6 Universality

There is currently a great deal of emphasis in empirical lossless data compression research upon how well given techniques perform on the Calgary Corpus. However, the more important and ultimately useful aspect of the techniques we present here is their principled, semantically coherent design. The modeling algorithms developed in this paper perform better on the benchmarks than existing algorithms do because our algorithms impose fewer and less restrictive assumptions on the input data.

For the following reasons, we believe that the improvements presented here are *universal*—that is, they will induce similar relative performance increases for most suffix-tree techniques on most data:

- The improvements we made are not *ad hoc* or empirically tuned to fit the Calgary Corpus.
- We showed the improvements to be independent of a key input parameter, the order bound, that can be empirically tuned to fit the Corpus.

Table 9.15: GDMC with owner protection, update exclusion X_1 and state selection.

	PS_2		PS_3		PS_5		PS_6	
File	bpc	select order	bpc	select order	bpc	select order	bpc	select order
bib	2.043	5.21	2.038	4.40	2.170	3.56	**2.029**	5.48
book1	2.363	6.30	**2.272**	3.80	2.292	3.47	2.319	5.37
book2	2.044	6.22	**2.016**	4.57	2.052	4.03	2.023	6.02
geo	5.303	3.40	**5.007**	2.06	5.016	1.88	5.033	2.23
news	2.622	5.48	2.606	4.34	2.633	3.90	**2.597**	5.48
obj1	4.402	16.36	**4.378**	15.64	4.455	13.91	4.394	35.37
obj2	2.767	5.32	2.776	4.85	2.896	3.97	**2.764**	5.30
paper1	2.578	4.67	2.574	3.69	2.593	3.43	**2.565**	4.32
paper2	2.449	5.07	**2.415**	3.53	2.437	3.23	2.423	4.52
pic	0.907	727.27	0.813	15.00	0.828	12.00	**0.794**	1619.95
progc	2.643	4.45	2.657	4.07	2.689	3.64	**2.638**	4.32
progl	1.837	6.06	1.847	5.13	1.921	4.19	**1.830**	6.46
progp	1.884	5.55	1.897	4.98	1.933	4.24	**1.880**	5.81
trans	1.719	5.68	1.722	5.30	1.784	4.45	**1.709**	6.65
Avg.	2.540		2.501		2.550		**2.500**	

- The changes improved the compression for each file individually, and there is considerable variety among the files of the Calgary Corpus.

Thus the improvements we have presented combine with PPM*—the first on-line stochastic model to impose no order assumptions and no arbitrary model growth heuristics on the data—to form what may be the most universal on-line modeling technique that has been evaluated empirically to date.

Chapter 10

CONCLUSION

In this dissertation I addressed the question of whether finite-order FSM models used in universal on-line max-likelihood modeling can be pushed much farther than they have already been pushed by other information theorists and data compression experimentalists. I explored the opportunity for improvement presented by a gap that remained between the theorists' goal of asymptotic convergence, and the experimentalists' goal of good predictions based upon typically small sample sizes. I identified and delineated this gap by introducing an orthogonal on-line model design framework and formally reducing several known techniques into the framework. I then filled the gap with the successful combination of convergent state selection and the construction of dynamically inherited frequency distributions that have the greatest weight when local frequency distributions are most uncertain. This chapter summarizes the contributions of this dissertation and then outlines a number of future projects.

10.1 Itemized Contributions

10.1.1 DMC Analysis

The DMC algorithm was introduced in 1986. Even in its original un-tuned form, its performance compared with carefully tuned state-of-the-art implementations of PPM, which has since been improved. However, since its introduction, the DMC algorithm has accumulated open questions rather than improvements or increased application. It is no coincidence that DMC is also the only FSM technique in the practical or theoretical literature that was not designed with an explicit context-based semantics. Chapter 4 proved a minimal, finite-context semantics for DMC, and Chapter 9 improved its performance.

DMC's Open Questions Answered

- *How does DMC's model structure partition the set of conditioning contexts?*

  - DMC's context partition can only be partially described by mapping any single finite conditioning context (string) to each state (Section4.3.1), but it does have a closed form (Theorem 4.4.1).
  - **DMC is provably not FSMX.** Therefore there is a fundamental difference in the families of languages that are accepted by PPM models and DMC models (Section 4.5.3). Incidentally, this also clears up the pervasive misconception that the FSMX class properly contains finite-order Markov models (Section 2.5).

- *How can we generalize DMC to larger alphabets without worsening compression performance or memory consumption?*
  - *Lazy cloning* with *inherit before novel event updates.*[1] (See section 6.5.)
- *How can we correct the exhaustive memory consumption that has kept the otherwise competitive DMC modeling algorithm from being used in practice?*
  - *Lazy cloning* as above, plus *owner protection*, which prevents transitions from being redirected, reduced DMC's memory requirements to those of PPM variants with similar performance (Section 9.5).

10.1.2 Unification of Prior-Art Techniques

A study of the practical data compression and related information theoretic literature may lead some scientists to this remarkable discovery: the science of reducing the redundancy of data could itself use some redundancy reduction. Chapter 8's canonical form for on-line modeling algorithms remedies this situation by exposing the abstract differences among existing and future techniques, once they are transformed. Furthermore, it also exposes the opportunities for genuine innovation that are explored in Chapter 9.

- The unifying framework enables meaningful, component-wise comparison among theoretical and practical techniques.

---

[1] I only claim "independent invention" of *lazy cloning*, and do so with the explicit agreement of the other inventors, Jukka Teuhola and Timo Raita, who published "GDMC" (which implements both concepts) first in [TR93]. In August 92, I reported the same ideas by mail to one of DMC's authors, Nigel Horspool, who invited me to present it in person at the University of Victoria in March 1993. We both were unaware of the independent invention.

- Orthogonality of components implies that the entire framework is *implementable* as the cross-product of unique features of known algorithms.
- Several novel, abstractly distinct algorithms therefore result from the implementation of the framework.
- The implemented framework provides a test bed for precisely controlled experiments with individual components and with component combinations.
- We were able to give performance results for previously existing techniques that had never been empirically evaluated.

10.1.3 A Bridge Between Sequential Coding Theory and Data Compression Practice

Information-theoretic on-line modeling algorithms stress asymptotic convergence and rely on local order estimation. Meanwhile, practical on-line modeling algorithms rely upon careful probability estimation to better utilize the scarce message statistics gathered from the relatively small files encountered in practice. These two approaches have been developed independently, are completely distinct, and have not been compared, either apart or in combination, until now.

Original Observations

- Information-theoretic algorithms simulate multiple context partitions, while practical algorithms progressively refine a single context partition.[2]
- All competitive, practical algorithms aggressively refine a single context partition.
- Information-theoretic algorithms hedge against over-refinement by using state selection to ignore refinements until their frequency distributions converge.
- Practical algorithms hedge against over-refinement with dynamically inherited frequency distributions that accelerate the convergence of a refinement's frequency distribution.
- All existing information-theoretic state selection methods rely upon suboptimal hill-climbing or order-bounds.

---

[2] This explains long-established empirical evidence that the use of multi-model frequency update (*full update*) techniques degrades the performance of PPM, which has single-model semantics.

Hypotheses

- State-selection *should* be combined with dynamically inherited frequency distributions so that more refined context-partitions will be selected earlier.
- State-selection *can* be combined with dynamically inherited frequency distributions if the computation of inheritances does not violate the multi-model semantics required for properly executing state selection, or if separate frequency distributions (one for each model semantics) are maintained.

Controlled Empirical Evaluations

One goal of our experiments was to provide conclusive data for model designers that shows the effects on compression performance and model size that each optimization has. We showed the following:

- The best-performing state selector is the percolating technique introduced in this thesis (Tables 9.3 and 9.4).
- The best-performing inheritance time is *inherit before novel updates*, which was introduced in this thesis (Table 9.5).
- The best-performing combination overall is a high-performance mixture, update exclusion, and percolating state selection (Tables 9.7 and 9.9).
- The best mixtures improve performance more that the best state selection (Table 9.5).
- Update exclusions improve performance nearly as much as the best state selector (Table 9.6).

In addition, we undertook improving the performance and memory consumption of three baseline data compression algorithms:

- We improved PPM*'s performance by 7% (Table 9.9).
- Our string-transition suffix-tree implementation reduced PPM*'s model size by half (Section 3.5).
- We improved PPM's performance by 12% over the standard reference [Mof90], and by 5% over the best PPM variant consisting of technologies available prior to this thesis (Tables 9.2 and 9.8).

- Our *order-bounded* string-transition suffix-tree implementation reduced PPM's model size extensively for higher order models, which were previously impossible to execute (Table 9.11).

- Our lazy cloning implementation, LazyDMC, plus "owner protection" and the optimizations we applied to PPM, improved DMC's performance by 12% over the original implementation, and by 5.6% over another independently derived lazy implementation, GDMC (Table 9.14).

- Our lazy cloning implementation, LazyDMC, uses roughly half of the memory that GDMC requires (Table 9.12).

10.2 Yet Another On-line Linear-Space Suffix-Tree Construction Algorithm

Chapter 3's original transformation of PPM* and PPM to a suffix-tree implementation with a minimal number of nodes was necessary to add PPM* to our taxonomy. The transformation describes an on-line linear-space suffix-tree construction algorithm. We started the work late in 1994, upon receiving a preprint of [CTW95]. However, Ukkonen published an on-line suffix-tree construction algorithm [Ukk95] first, in 1995, while we were combining ours with state selection and mixtures in the implementation of our taxonomy. The independently derived algorithms are quite similar—even down to the use of the term "splitting." Both algorithms are functionally related to McCreight's suffix-tree construction algorithm [McC76].

Additionally, Larsson [Lar96] applied Ukkonen's construction to PPM* and described how to make the suffix-tree represent a sliding window of input history. Thus, our on-line suffix-tree construction algorithm is not the first, nor is our application of on-line suffix-tree construction algorithms to implementing PPM*. However, our application of a linear-space on-line suffix-tree model constructor to order-bounded (PPM) models and to suffix-tree models with mixtures or state selection, are the first, and required non-trivial solutions. Furthermore, ours is the first actual implementation of PPM and PPM* using the construction that we know of.

10.3 Future Directions

This thesis explored the question of what level of prediction performance is possible with certain classes on-line FSM models, regardless of cost. There are a number of ways that we can use these results to further understand the problem of modeling sequences, as well as to build better practical modeling algorithms. Three straightforward directions involve *completing* the study of DMC and the class of models that can be constructed via edge-redirection, *applying* the results from our analyses and empirical studies, and *extending* the taxonomy to include a recent theoretical technique that seems to be quite distinct from the approaches already covered.

Complete our study of DMC and finite-order FSM model classes:

- Completely describe the relationship of models grown via edge-redirection (DMC models) to the class of Finite-Order Languages. Is DMC "complete" for the class?
- We proved that DMC models are more powerful than (i.e., properly include) FSMX models, but failed to discover a (non-FSMX) DMC model that outperformed the best FSMX model. Future results should either explain why this cannot be done, or demonstrably harness DMC's extra power. Doing so will require growing models more slowly than PPM and PPM* do, which degrades performance on small files. If there is a way to harness DMC's extra power, it will probably be demonstrable only with larger files.
- In this thesis we fairly conclusively answered the questions, *What is the best state selector*, *What is the best update mechanism*, and *What is the best existing mixture?* However, we did not explore the question, *What is the best combination of edge-redirection structure and criterion?* Performing the experiments that will answer this question could require exploring a more extensive list of edge-redirection criteria than we included in the taxonomy. This is also the path we would first investigate to demonstrably harness DMC's extra power.

Apply the best-performing approaches identified in this thesis:

- This work can be used to take some of the "ready, fire, aim!" out of empirical data compression research. The empirical studies in this thesis, and the

hundreds of other empirical studies that are made possible by the executable taxonomy and future extensions of it, can be used to help model designers decide which computations to eliminate, approximate, or perform exactly, based upon known performance/design-time/resource tradeoffs.

- The modeling techniques presented and evaluated in this thesis can be applied to other on-line and off-line modeling problems such as speech recognition and genomics. For example, the lattice-structured Hidden Markov Models (HMMs) [Rab89] that are commonly used in speech and genomics model position-related data dependencies explicitly, but not context-related dependencies. Thus, speech and genomic sequences are currently modeled with zero-order statistics. We propose that a hierarchically designed model that will utilize both types of dependencies can be constructed by replacing the *states* of HMMs with the context models studied in our work (complete with state selection), without greatly affecting the computational complexity of the dynamic-programming algorithms that are used to parameterize the HMM. The resulting FSM is a variable-order HMM.

Extend the taxonomy:

- Recently, a new theoretical on-line modeling technique called "The Context-Tree Weighting Method" was published in [WST95]. It explicitly constructs binary FSMX models, but it very explicitly does not perform state selection. How does the Context-Tree Weighting Method fit into our framework, or more precisely, how must we alter the framework to incorporate it? How can we efficiently generalize the Context-Tree Weighting Method to an m-ary alphabet? How does it perform? Can we improve it? How do the new subproblem solutions that it contributes to the framework (at the very least, it adds an entropy-based mixture weighting function), perform relative to the others?

The executable taxonomy is scheduled to be made publicly available in mid to late 1997, at the on-line host `http://www.cs.washington.edu`, for use and extension by other researchers.

Bibliography

[Ash65] R. Ash. *Information Theory.* John Wiley and Sons, New York, New York, 1965.

[BB92] S. Bunton and G. Borriello. Practical dictionary management for hardware data compression. *Communications of the ACM*, 35(1):95–104, 1992.

[BCW90] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Advanced Reference Series. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Bel86] T. C. Bell. *A Unifying Theory and Improvements for Existing Approaches to Text Compression.* PhD thesis, University of Canterbury, 1986.

[BM89] T. C. Bell and A. Moffat. A note on the DMC data compression scheme. *The British Computer Journal*, 32(1):16–20, 1989.

[Bun92] S. Bunton. Data structure management tagging system. U.S. Patent 5,151,697, September 1992.

[Bun94] S. Bunton. A characterization of the 'Dynamic Markov Compression' FSM with finite conditioning contexts. UW-CSE Technical Report UW-CSE-94-11-03, The University of Washington, November 1994.

[Bun96] S. Bunton. *On-Line Stochastic Processes in Data Compression.* PhD thesis, University of Washington, December 1996.

[Bun97a] S. Bunton. A percolating state selector for suffix-tree context models. In *Proceedings Data Compression Conference.* IEEE Computer Society Press, March 1997.

[Bun97b] S. Bunton. Semantically motivated improvements for PPM variants. *The British Computer Journal, Special Data Compression Issue*, 1997. (invited paper, to appear June 1997).

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, DEC SRC, May 1994.

[BWC89] T. C. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computer Surveys*, 24(4):555–591, 1989.

[CH87] G. V. Cormack and R. N. S. Horspool. Data compression using dynamic Markov modelling. *The Computer Journal*, 30(6):541–550, 1987.

[Cho88] P. A. Chou. *Applications of Information Theory to Pattern Recognition and the Design of Decision Trees and Trellises.* PhD thesis, Stanford University, 1988.

[CKW91] D. Chevion, E. D. Karnin, and E. Wallach. High efficiency, multiplication-free approximation of arithmetic coding. In *Proceedings Data Compression Conference*, pages 43–52. IEEE Computer Society Press, March 1991.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley and Sons, New York, New York, 1991.

[CTW95] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. In *Proceedings Data Compression Conference*, March 1995.

[CW84a] J. G. Cleary and I. H. Witten. A comparison of enumerative and adaptive codes. *IEEE Transactions on Information Theory*, 30(2):306–315, 1984.

[CW84b] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[FG89] E. Fiala and D. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, April 1989.

[FGC93] G. Feygin, P. G. Gulak, and P. Chow. Minimizing error and VLSI complexity in the multiplication-free approximation of arithmetic coding. In *Proceedings Data Compression Conference*, pages 118–124. IEEE Computer Society Press, March 1993.

[Fur91] G. Furlan. An enhancement to universal modeling algorithm 'context' for real-time applications to image compression. In *IEEE Transactions on Acoustics Speech and Signal Processing*, pages 2777–2780, 1991.

[Gua80] M. Guazzo. A general minimum-redundancy source-coding algorithm. *IEEE Transactions on Information Theory*, 26(1):15–25, January 1980.

[How93] P. G. Howard. *The Design and Analysis of Efficient Lossless Data Compression Systems*. PhD thesis, Brown University, 1993.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[HV92] P. G. Howard and J. S. Vitter. Practical implementations of arithmetic coding. In J. A. Storer, editor, *Image and Text Compression*, pages 85–112. Kluwer Academic Publishers, Norwell Massachusetts, 1992.

[Lan83] G. G. Langdon. A note on the Ziv-Lempel model for compressing individual sequences. *IEEE Transactions on Information Theory*, 29(2):284–287, 1983.

[Lar96] N. J. Larsson. Extended application of suffix trees to data compression. In *Proceedings Data Compression Conference*, pages 190–199, March 1996.

[LR83] G. G. Langdon and J. J. Rissanen. A double-adaptive file compression algorithm. *IEEE Transactions on Communications*, 31(11):1253–1255, 1983.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[MGZ89] N. Merhav, M. Gutman, and J. Ziv. On the estimation of the order of a Markov chain and universal data compression. *IEEE Transactions on Information Theory*, 35(5):1014–1019, 1989.

[Mof90] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990.

[MSWB93] A. Moffat, N. Sharman, I. H. Witten, and T. C. Bell. An empirical evaluation of coding methods for multi-symbol alphabets. In *Proceedings Data Compression Conference*, pages 108–117, March 1993.

[MW85] V. S. Miller and M. N. Wegman. Variations on a theme by Ziv and Lempel. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms*

on Words, pages 131–140. Springer-Verlag, New York, New, York, 1985.

[Pas76] R. Pasco. *Source Coding Algorithms for Fast Data Compression.* PhD thesis, Stanford University, 1976.

[Paz71] A. Paz. *Introduction to Probabilistic Automata.* Academic Press, 111 Fifth Avenue, New York, New York 10003, 1971.

[Rab89] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. IEEE*, 77(2):257–286, 1989.

[RC89] T. V. Ramabadran and D. L. Cohn. An adaptive algorithm for the compression of computer data. *IEEE Transactions on Communications*, 37(4):317–324, 1989.

[Ris76] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, May 1976.

[Ris83] J. J. Rissanen. A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–664, 1983.

[Ris86a] J. J. Rissanen. Complexity of strings in the class of Markov sources. *IEEE Transactions on Information Theory*, 32(4):526–532, 1986.

[Ris86b] J. J. Rissanen. An image compression system. In *Proceedings HILCOM 86*, 1986.

[Ris89] J. J. Rissanen. *Stochastic Complexity in Statistical Inquiry.* World Scientific Publishing, Singapore, 1989.

[Ris90] E. A. Riskin. *Variable-Rate Vector Quantization of Images.* PhD thesis, Stanford University, 1990.

[RL79] J. J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23:146–162, March 1979.

[RL81] J. J. Rissanen and G. G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, 27(1):12–23, 1981.

[RM89] J. J. Rissanen and K. M. Mohiuddin. A multiplication-free multialphabet arithmetic code. *IEEE Transactions on Communications*, 37:93–98, 1989.

[Rub79] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Transactions on Information Theory*, 25:672–675, November 1979.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, March 1948.

[Sto85] J. Storer. Textual substitution techniques for data compression. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 111–129. Springer-Verlag, 1985.

[Ton93] T. Y. Tong. *Data Compression with Arithmetic Coding and Source Modeling*. PhD thesis, University of Waterloo, 1993.

[TR93] J. Teuhola and T. Raita. Application of a finite-state model to text compression. *The Computer Journal*, 36(7):607–614, 1993.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[vL90] J. van Leeuwen. *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*. Advanced Reference Series. The MIT Press, 55 Hayward Street, Cambridge MA 02142, 1990.

[WB91] I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, 1991.

[Wel84] T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[Whi94] R. F. Whitehead. An exploration of dynamic Markov compression. Master's thesis, University of Canterbury, 1994.

[Wil91] R. N. Williams. *Adaptive Data Compression*. Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[WLZ92] M. J. Weinberger, A. Lempel, and J. Ziv. A sequential algorithm for the universal coding of finite memory sources. *IEEE Transactions on Information Theory*, 38(3):1002–1014, 1992.

[WMB94] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, New York, 1994.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[WST95] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens. The context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41(3):653–664, 1995.

[Yu93] T. L. Yu. Hybrid dynamic Markov modeling. In *Proceedings Data Compression Conference*. IEEE Computer Society Press, March 1993.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

Suzanne Bunton was born in Lubbock, Texas during the 1960s. She completed her Bachelor of Science degree in Mathematics, *magna cum laude*, at the University of Texas at Dallas in 1987. She received a Master's degree in Computer Science from the University of Washington in 1990. Portions of her master's thesis, "Practical Dictionary Management for Hardware Data Compression," were published in the 1990 MIT VLSI Conference Proceedings, the January 1992 issue of The Communications of the ACM, and in U.S. Patent 5,151,697. She finished her Ph.D. dissertation, "Stochastic Processes in Data Compression," in the Department of Computer Science and Engineering at the University of Washington in 1996. Parts of the thesis were published in the 1995 and 1997 Data Compression Conference Proceedings, and in an invited submission for the 1997 Special Data Compression Issue of The British Computer Journal. After receiving her Ph.D., she accepted a postdoctoral fellowship in the Department of Molecular Biotechnology at the University of Washington, where she will apply stochastic modeling techniques to problems involving genomic sequences.