Linear Time Universal Coding and Time Reversal of Tree Sources via FSM Closure

Alvaro Martin, Gadiel Seroussi, Marcelo J. Weinberger
Information Theory Research
HP Laboratories Palo Alto
HPL-2003-87(R.2)
March 15, 2004*

tree sources, finite state machines, finite memory, Context algorithm, universal coding, suffix trees

Tree models are efficient parametrizations of finite-memory processes, offering potentially significant model cost savings. The information theory literature has focused mostly on redundancy aspects of the universal estimation and coding of these models. In this paper, we investigate representations and supporting data structures for finite-memory processes, as well as the major impact these structures have on the computational complexity of the universal algorithms in which they are used. We first generalize the class of tree models, and then define and investigate the properties of the finite state machine (FSM) closure of a tree, which is the smallest FSM that generates all the processes generated by the tree. The interaction between FSM closures, generalized context trees, and classical data structures such as compact suffix trees brings together the information-theoretic and the computational aspects, leading to an implementation in linear encoding/ decoding time of the semi-predictive approach to the Context algorithm, a lossless universal coding scheme in the class of tree models. An optimal context selection rule and the corresponding context transitions are computationally not more expensive than the various steps involved in the implementation of the Burrows-Wheeler transform (BWT) and use, in fact, similar tools. We also present a reversible transform that displays the same "context deinterleaving" feature as the BWT but is naturally based on an optimal context tree. FSM closures are also applied to an investigation of the effect of time reversal on tree models, motivated in part by the following question: When compressing a data sequence using a universal scheme in the class of tree models, can it make a difference whether we read the sequence from left to right or from right to left? Given a tree model of a process, we show constructively that the number of states in the tree model corresponding to the reversed process might be, in the extreme case, quadratic in the number of states of the original tree. This result answers the above motivating question in the affirmative.

* Internal Accession Date Only
To be published in IEEE Transactions on Information Theory
Approved for External Publication
© Copyright IEEE

Linear Time Universal Coding and Time Reversal of Tree Sources via FSM Closure*

Alvaro Martín[†] Gadiel Seroussi[‡] Marcelo J. Weinberger[‡]

Abstract

Tree models are efficient parametrizations of finite-memory processes, offering potentially significant model cost savings. The information theory literature has focused mostly on redundancy aspects of the universal estimation and coding of these models. In this paper, we investigate representations and supporting data structures for finite-memory processes, as well as the major impact these structures have on the computational complexity of the universal algorithms in which they are used. We first generalize the class of tree models, and then define and investigate the properties of the finite state machine (FSM) closure of a tree, which is the smallest FSM that generates all the processes generated by the tree. The interaction between FSM closures, generalized context trees, and classical data structures such as compact suffix trees brings together the information-theoretic and the computational aspects, leading to an implementation in linear encoding/decoding time of the semi-predictive approach to the Context algorithm, a lossless universal coding scheme in the class of tree models. An optimal context selection rule and the corresponding context transitions are computationally not more expensive than the various steps involved in the implementation of the Burrows-Wheeler transform (BWT) and use, in fact, similar tools. We also present a reversible transform that displays the same "context deinterleaving" feature as the BWT but is naturally based on an optimal context tree. FSM closures are also applied to an investigation of the effect of time reversal on tree models, motivated in part by the following question: When compressing a data sequence using a universal scheme in the class of tree models, can it make a difference whether we read the sequence from left to right or from right to left? Given a tree model of a process, we show constructively that the number of states in the tree model corresponding to the reversed process might be, in the extreme case, quadratic in the number of states of the original tree. This result answers the above motivating question in the affirmative.

Keywords: Context algorithm, Finite memory, Finite state machines, Suffix trees, Tree sources, Universal coding.

*Parts of this paper evolved from material presented at the 1995 International Symposium on Information Theory, Whistler, British Columbia, Canada. Parts were also presented at the 2004 Data Compression Conference, Snowbird, Utah, USA.

[†]Instituto de Computación, Universidad de la República, Montevideo, Uruguay. The work of this author was done while he was, on leave, at Hewlett-Packard Laboratories, Palo Alto, California, and was partially supported by the Office of Naval Research, Grant N00014-03-1-0399.

[‡]Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, USA.

1 Introduction

In a *finite-memory* process, the conditional probability assigned to the next emitted symbol, given all the past, depends only on a finite number m of contiguous past observations. This class of processes can be parametrized with a Markov model of order m, but since for practical data the actual memory length often varies from location to location, such parametrizations can be very inefficient. The number of model parameters, which grows exponentially with m in a Markov model, can be dramatically reduced by lumping together equivalent states (i.e., m-vectors) that yield identical conditional distributions. In lossless coding, for example, this reduction can improve the rate at which the average length of a *universal* code can converge to the entropy for most parameter values, as Rissanen's lower bound [1, Theorem 1] on this average includes a *model cost* term proportional to the number of parameters. The reduced models, first considered in [2], were termed *tree models* in [3], since they can be represented with a simple tree structure. Roughly speaking, a tree model consists of a full α-ary context tree,[1] where α is the size of the source alphabet, and a set of conditional probability distributions on the alphabet, one associated with each leaf of the tree (the *states*). Tree models have also been adopted as data models in statistics (being referred also as *variable length Markov chains* [4] in the literature). The appeal of this class of models is two-fold: on one hand, it appears to efficiently capture redundancies typical of real life data (e.g., text or images), while on the other hand the models in the class can be optimally estimated using the *Context* algorithm in its various flavors, e.g. [2, 3, 5]. Moreover, the *Context Tree Weighting* (CTW) algorithm [6, 7] produces a sequential probability assignment which is a two-stage mixture of all models in the class. The appeal of such two-stage mixtures was first observed in [8], leading in the lossless data compression application to a coding scheme which is universal also in the setting of individual sequences.

In this work, we investigate various representations of finite-memory processes, as well as supporting data structures and their impact on the computational complexity of the algorithms in which they are applied. In contrast to this algorithmic approach, most of the discussion of tree models in the information-theoretic literature has focused on the redundancy aspects of their universal modeling and coding, be it with the "plug-in" type of approach of the Context algorithm [2, 9, 3], with the mixture approach of the CTW algorithm [6, 7], or with the *two-pass* approach outlined in [10] for countable "hierarchies" of models. One major application of our results is an algorithm that implements, in linear encoding/decoding time, a classical universal code derived from the latter approach. Since this application highlights the importance of data structures in the analysis of tree models, we next discuss in more detail the literature on universal lossless coding for these models, as well as other popular, non-universal coding schemes that also employ tree models.

The redundancy of universal lossless codes for tree models is analyzed in the framework of *double universality* [10, 11], in which it is shown that for any tree model with K free parameters, the normalized excess code length given by these codes on sequences of length n, over the empirical entropy determined by the model, is at most $(K \log n)/(2n)+O(K/n)$, for any K. This upper bound holds for any individual sequence with the mixture and two-pass approaches, or in the average (i.e., when the reference model is assumed to have indeed produced the data) with the plug-in approach. Thus, optimality for *most*

[1] We say that an α-ary tree T is *full* if every internal node of T has exactly α children; T is *full balanced* if it is full and all its leaves are at the same depth.

parameter values and any model size in a probabilistic setting follows from Rissanen's lower bound [1] (see [12] for a notion of optimality for "most" sequences in a deterministic setting). In some works, a bound on the value of K is assumed [9, 5, 6]; this assumption is removed from the analysis in [3] for the plug-in approach, and in [7] for the mixture approach (see also [12] for a broader model class). Since much of the emphasis is on sequentiality, the two-pass approach of [10], in which the best model structure in a hierarchy (e.g., a tree) is estimated and described to the decoder in a first pass, and then the data is encoded in a second pass with a universal code for the above best model structure, has not received much attention. In [13], this approach is termed "semi-predictive" for the case in which the universal code used for a given model structure is sequential. The semi-predictive approach for the class of tree models (see, e.g., [5, 14]), competes with CTW (as, with an appropriate probability assignment [15], they both achieve double universality for individual sequences), but it is redundant in the sense that once the best tree is described, coding space is still allocated to sequences for which this tree is not optimal. The mixing approach overcomes this intrinsic redundancy, and is therefore preferred in theory, although the per-symbol difference in code length is clearly of order $O(\hat{K}/n)$, where $\hat{K}$ denotes the number of leaves in the tree that yields the shortest code in the second pass.[2]

A major advantage of tree models (over, e.g., models based on general finite-state machines) is that the statistical information needed to implement the above schemes can be stored in a context tree, which is grown as the sequence is observed, recording essentially all the occurrences of each letter in every context. The manner in which context trees are employed depends on the coding approach. With the plug-in approach, a "distinguished" coding context is sequentially selected for each symbol to be coded (context selection rule), and the coding distribution is conditioned on this context. With the semi-predictive approach, the context tree is "pruned" to minimize code length [5], and described to the decoder in a first pass through the data. In a second pass, again, each symbol is assigned a conditional probability sequentially, conditioned on the coding context determined by the pruned tree (which is used as a state). This assignment, in turn, is used for, e.g., arithmetic coding. With the mixing approach, the probability assignment is a mixture of assignments for all possible contexts. Since the literature did not focus on the computational complexity of these algorithms, the basic data structure to describe tree models and context trees has typically been a plain *atomic tree* [16]. It is only recently that more efficient data structures such as *compact suffix trees* [16] have been discussed in connection with these algorithms [17].

On the other hand, the use of suffix trees is customary in the implementation of popular data compression algorithms that are also based on context models but lack the above strong universality properties, such as PPM [18] in its multiple variants, and those based on the Burrows-Wheeler transform (BWT) [19] (see [20] and references therein). Moreover, suffix trees are crucial for low complexity implementations [21, 20]. The approach in PPM is similar to the context algorithms of the plug-in type, except that the context selection rule is a heuristic based on the number of occurrences of the possible contexts. The BWT-based algorithms can also be viewed as coding based on a context tree (see [22] for an information-theoretic analysis), except that no attempt at context selection is made. Instead, the sequence is reordered based on a traversal of the context tree such that symbols occurring in "similar"

[2] In a probabilistic setting in which the data is assumed to be drawn from a tree source with K^* parameters, $\hat{K} = K^*$ with high probability, due to the consistency of the Minimum Description Length estimator.

contexts appear in nearby locations. Coding is often done by sub-optimal, simple methods in a second pass. While these works emphasize data structures and complexity analysis, the focus on redundancy rates for the universal context-based codes relegated the investigation of algorithmic aspects. The reader is referred to [17] for a study of the computational complexity of proposed implementations of these universal codes, which demonstrates that no algorithm for both encoding and decoding in linear time is available. In the past, this dichotomy may have led to the misconception that the implementation of such codes is hopelessly complex. As a result, implementable context algorithms have often limited the tree depth [5, 6].

The popularity of BWT-based schemes suggests that, in many applications, sequentiality is not a fundamental requirement. Thus, in such cases, a low-complexity implementation of the semi-predictive approach is of interest, despite the outlined slight theoretical disadvantage relative to mixing schemes. Notice that this approach is especially suited for a clean complexity analysis, as three major issues can be identified and treated separately:

(a) Gathering of all relevant statistical information in a context tree;

(b) Pruning of the tree at the encoder, to obtain the model that minimizes the code length; and

(c) Transitioning from context to context.

The "relevant statistical information" mentioned in (a) is different at the encoder and the decoder, since, at the encoder, it must facilitate the optimization procedure in (b), which is not needed at the decoder. While compact suffix trees address, as we discuss later, some of the computational issues in (a)–(b), the tree structure of the model is clearly an obstacle for transitioning from context to context in a constant number of operations per symbol, as it requires descending the (pruned) tree starting at the root until the new context is found. This context may occur at a depth that is not necessarily bounded by any constant independent of n. In this sense, a finite-state machine (FSM) is preferable, since fast transition between states is built into the model definition. However, FSMs do not enjoy the hierarchical data collection advantages of trees. Unfortunately, as noted in [9], a minimal tree model might not be representable as an FSM with the same number of states. Thus, further research into these data structures is necessary for an efficient implementation of the three computations above.

Since the relevant statistical information can be organized in the compact suffix tree of the given string, the classical algorithmic tools surveyed in [16] are instrumental in addressing the first issue above in linear time.[3] In fact, the techniques in [20], while targeted at PPM, imply that suffix trees are instrumental for any scheme based on tree models (see also [21]). As for the second issue, [5] showed that, due to the (full) tree structure of the model class, pruning reduces essentially to a dynamic programming problem, with the cost function given by the code length that each potential node in the context tree would contribute in case it were selected as a state. This problem can also be solved in time that is linear in the number of nodes of the context tree, which, with a compact representation of the suffix tree, can in turn be made linear in the sequence length. Combining [23] and [5], and addressing the third issue by use of the BWT, Baron and Bresler [17] recently showed that the semi-predictive

[3]Throughout, we will measure complexity by the number of *register-level* operations, defined as arithmetic and logic operations, address computations, and memory references, on operands of size $O(\log n)$.

approach can be implemented in linear *encoding* time.[4] Unfortunately, the BWT is not available during decoding to provide a constant transition time per symbol.[5]

In this paper, we first formalize an extension of the class of tree models, letting the model structure take the form of a *compact digital tree*. In the extended model, the trees need not be full (so that states may be given by nodes other than the leaves), and the edges may be compacted (i.e., labeled by strings of length greater than one). This extension serves here as an auxiliary structure, facilitating the use of suffix trees which are generally not full (a key factor in maintaining linear time complexity). However, we formalize and discuss the class of generalized tree models in detail on its own right, as this richer class offers potentially significant improvements in model fitting capability relative to the usual full-tree models. The derivation of efficient algorithms to capitalize on these potential savings, however, remains an open problem of both theoretical and practical interest. We then proceed to define the FSM *closure* of a tree, which is the smallest FSM that generates all the processes generated by the tree as the parameters are allowed to range over their valid domain. We present an algorithm that builds this closure in time that, in case the tree was derived by optimal pruning of a suffix tree, is *linear* in the length of the sequence that generated the suffix tree. Again, our formalization of the concept of FSM closure and the study of its properties extends beyond the applications considered in this paper.

In the first such application, generalized trees and their FSM closures allow us to achieve linear time encoding/decoding complexity for the semi-predictive twice-universal code in the class of full-tree models, without recourse to the BWT, by solving the context transition problem efficiently. We point out that, in this application, our contribution is algorithmic in nature, in the sense that, as discussed, the proposed algorithm implements a code whose double universality is well known. Our algorithm shows that the key complexity issues pertain to data structures, and that a judicious choice of these structures can be done for universal, context-based schemes, as efficiently as with the sub-optimal approach of coding based on the BWT. An optimal context selection rule, and the corresponding context transitions, are computationally not more expensive than the various steps involved in the implementation of BWT-based coding schemes. Furthermore, we present a reversible transform that displays the same "context deinterleaving" feature as the BWT but is naturally based on an optimal context tree. The comparison leads to the observation that the claimed advantages of BWT result just from a clever use of compact suffix trees, even for variants that rule out the use of arithmetic coding. The proposed transform is related to work in [23].

In a second application, we use the FSM closure to investigate the effects of *time reversal* on the structure of the minimal tree model of a finite-memory process. This problem is motivated in part by the following simple question that arises in some data compression applications: When compressing a data sequence with a twice-universal code in the class of tree models, can it make a difference whether we read the sequence from left to right or from right to left? Time reversal of stationary Markov processes is well understood in the literature. In particular, it is known that time reversal preserves

[4]To complete a linear time encoder, it is shown in [17] that the cost functions used in the pruning step can be computed with the required precision on registers of size $O(\log n)$ in overall linear time, whereas the extra redundancy due to arithmetic coding was shown in [9] to be negligible, provided again that the coding operations are carried out on registers of size $O(\log n)$.

[5]An alternative approach for efficient context transition, used for PPM in [20], is the use of suffix links. For the semi-predictive Context algorithm, again, this approach cannot be used directly at the decoder.

both the order and the entropy of a stationary Markov process (see, e.g., [24, Ch. 4]). This fact still leaves the question of the effect on the size of the minimal tree model (which is crucial in the setting of double universality) open. To address this question, we characterize the class of *two-sided finite-memory processes* whose time-reversed versions are well defined. These processes also admit tree models, and for a given tree T, we present a construction of the minimal tree that generates the reverses of all the processes generated by T. This "reverse" tree turns out to be linked to the FSM closure of T. We show that the number of states in the reverse tree might be, in the extreme case, quadratic in the number of states of T. This result yields an affirmative answer to the above motivating question.

The remainder of this paper is organized as follows. Section 2 formally defines the (extended) class of tree models and investigates its properties. Section 3 introduces the FSM closure of a tree, investigates bounds on its size, and presents a linear time algorithm for its construction. Section 4 applies the FSM closure and associated data structures to a linear time implementation of the semi-predictive universal code in the class of full-tree models. Section 5 introduces the concept of two-sided processes, and investigates the effect of time reversal on the size of a tree model. Finally, Section 6 concludes the paper.

2 Generalized context tree models

2-A Finite-memory processes and tree models

In this sub-section, we review finite-memory processes and their parametrizations, particularly tree models. While most concepts we discuss are not novel, some of the formalisms (e.g., the transient states of a tree) are. An important aspect emphasized in this review is the distinction between a process and its representations. We first introduce some notation. Let A be an alphabet of $\alpha \geq 2$ symbols, and let λ denote the empty string. As is customary, we let A^*, A^+, and A^m, denote, respectively, the set of finite strings, the set of positive-length strings, and the set of strings of length $m \geq 0$ over A. We use the notation u_j^k as shorthand for $u_j u_{j+1} \dots u_k$, $u_i \in A$, $j \leq i \leq k$, and extend it by defining $u_j^k = \lambda$ when $j > k$. Also, we omit the subscript when $j = 1$, i.e., $u^k = u_1^k$. For $u = u^k$, we let $|u| = k$ denote the length of u, $\overline{u} = u_k u_{k-1} \dots u_1$ its reverse string, $\mathrm{head}(u)$ its first symbol, u_1 (or λ if $k = 0$), and $\mathrm{tail}(u) = u_2^k$ its longest proper suffix. For strings $u, v \in A^*$, we denote by uv the concatenation of u and v. If u is a prefix (resp. proper prefix) of v, we write $u \preceq v$ (resp. $u \prec v$) or $v \succeq u$ (resp. $v \succ u$). Formally, we use the terms *string* and *sequence* interchangeably, but favor the latter term in cases where the sequence length is presumed to be unbounded.

Following [25], we consider a (probability assignment) function P from A^* into the real interval $[0, 1]$ satisfying the conditions

(Q1) $P(\lambda) = 1$,

(Q2) $P(u) = \sum_{a \in A} P(ua), \quad \forall u \in A^*$.

We will refer to P as a *string process*, or simply a *process* (the term *information source* is used in [25]). Notice that although the string process formalism is different from the usual setting of discrete time, discrete space random processes, all notions of interest in the conventional setting can

be expressed very naturally with string processes. For example, assuming $P(x^n) \neq 0$, the function $P(a|x^n) \triangleq P(x^n a)/P(x^n)$, $a \in A$, is a *conditional probability mass function* (CPMF) by **(Q2)**, and is naturally interpreted as the probability of the "next" symbol x_{n+1} being equal to a, conditioned on x^n.[6] The string process setting, on the other hand, is very natural when discussing universal coding schemes, which can be regarded as carefully crafted string processes [25].

One way of generating string processes is by use of a *recursive model* [25]. Specifically, given a set of *states* S, consider a *state function* $\sigma : A^* \to S$ and a set of CPMFs $\{p(\cdot|s)\}_{s\in S}$. For an arbitrary sequence $x^n \in A^n$, let the state sequence s_0^n be given by $s_i = \sigma(x^i)$, $0 \le i \le n$, and define the function P by

$$P(\lambda) = 1; \quad P(x^n) = \prod_{i=1}^{n} p(x_i|s_{i-1}), \ \ n \ge 1\,. \tag{1}$$

Clearly, this assignment defines a string process. We say that the model, denoted $\langle \sigma, p \rangle$, generates the process P.[7] For any state s, and x^n such that $\sigma(x^n) = s$, we say that x^n *selects* s, and that s *accepts* x^n. A state is called *permanent* if it accepts arbitrarily long sequences; otherwise, the state is called *transient*. An important particular class of state functions considered, e.g., in [26], is defined through *finite state machines*. For our purposes, an FSM over A is given by a triple $\mathcal{F} = (S, f, s_0)$, where S is a finite set of states, $f : S \times A \to S$ is a *next-state function*, and $s_0 \in S$ is the *initial state*. The state sequence s_0^n for x^n is recursively defined by $s_i = f(s_{i-1}, x_i)$. In classical probability theory, the state sequence corresponds to a Markov chain (cf., e.g., [27]). Notice, however, that our definition of permanent state is based solely on the state function, and is independent of the CPMFs associated with the states. Thus, this definition differs from the notion of a *recurrent* state in the theory of Markov chains, which depends on the CPMFs. It is possible to find CPMF assignments that will make a permanent state non-recurrent (provided that some conditional probabilities are set to zero). Our notion of permanent state corresponds to one for which there exists *some* assignment of CPMFs that makes the state recurrent in the classical sense.[8] Our transient states, on the other hand, are always non-recurrent in the classical sense, independently of the CPMFs. Notice that if $s' = f(s, a)$ and s is a permanent state then so must be s' (as it accepts strings of arbitrary length).

For a set of strings $B \subseteq A^*$, and a process P, we define $B_P = \{u \in B \,|\, P(u) \neq 0\,\}$. A process P has the *finite-memory property* if there exists a nonnegative integer m such that, for all $n \ge m$, $a \in A$, and $x^n \in A_P^n$, $P(a|x^n)$ satisfies

$$P(a|x^n) = P(a|x_{n-m+1}^n). \tag{2}$$

The minimum integer m for which the finite-memory property holds for P is referred to as the *order* of the process. Clearly, this property holds for m if P can be generated with a recursive model such that, for all $n \ge m$ and $x^n \in A^n$, x^n selects the same state as x_{n-m+1}^n.

Conversely, every finite-memory process P of order m can be generated by a "basic" FSM such that $S = \cup_{j=0}^{m} A^j$, $s_0 = \lambda$, and for $a \in A$ and $b_1^j \in A^j$, $0 \le j \le m$, the next-state function is given by

[6]When $P(x^n) = 0$, the numerator in the definition of $P(a|x^n)$ must also vanish by **(Q2)**, and the function is undefined.

[7]This model is termed *recursive* in [25] since, in full generality, σ is any recursive function on A^*.

[8]In fact, all but a set of measure zero of the assignments will make a permanent state recurrent. In this sense, the structural model properties we will be interested in will be generally graph-theoretic or algebraic, will be required to hold for "some choice" of CPMFs, but will actually hold "for most choices."

$f(b_1b_2\dots b_j, a) = b_1b_2\dots b_ja$ for $j < m$, and $f(b_1b_2\dots b_m, a) = b_2\dots b_ma$. To complete the FSM model it suffices to select $p(a|b_1b_2\dots b_j) = P(a|b_1^j)$ for all $b_1^j \in A_P^j$, $0 \le j \le m$ (the choices when $P(b_1^j) = 0$ are inconsequential). On the other hand, not all FSM-generated processes are finite-memory [26]. Notice that the states corresponding to strings shorter than m symbols in the above FSM are transient, and their sole purpose is to accommodate arbitrary CPMFs $P(\cdot|x^n)$ for $n < m$ (these CPMFs are not constrained by (2)). As an alternative to transient states, a *particular* assignment for these strings is often obtained by letting $S = A^m$ and assuming that s_0 is a given fixed state. In any case, for a given finite set S and arbitrary n, the computation in (1) involves a constant number of factors $p(x_i|s)$ for transient states s (and each transient state occurs at most once). Thus, the contribution of transient states to the ideal code length, $-\log P(x^n)$, is $O(1)$, and the properties of the process of most interest to us are determined by the permanent states (each carrying, in general, $\alpha-1$ parameters). Transient states are just a "nuisance" that requires formal treatment, but has no impact on the main results.

The finite-memory property depends only on the probability assigned to long enough sequences. To simplify the discussion, we will also constrain the choice of probabilities conditioned on short sequences by further requiring, for each string $v \in A^*$, the condition

$$\text{if } P(a|uv) \text{ is independent of } u \ \forall u \in A_P^+, \text{ then } P(a|uv) = P(a|v), \quad a \in A \tag{3}$$

on the process P. The condition requires that probability assignments conditioned on short strings be consistent with the memory properties of longer strings, ruling out situations, for example, in which the order of the process is determined by the CPMFs of the transient states.[9]

For each particular finite-memory process of order m, other FSM representations may involve less than α^m (permanent) states. A *tree model* (see, e.g., [2, 9, 3]) is another type of recursive model (possibly not an FSM) which may involve less states than the above "basic" FSM. In a tree model, the permanent states are not necessarily all of the same length m. Specifically, given a full prefix-free set over A^*, the state selected by x^n is given by the (unique) prefix of $\overline{x}^n$ in the set, if n is large enough for such a prefix to exist (permanent state), or by $\overline{x}^n$ otherwise (transient state). Thus, the set of permanent states is most naturally represented by the leaves of a full α-ary tree. We will represent the states as strings $\sigma(x^n) = x_nx_{n-1}\dots x_{n-j}$, where the symbols are reversed relative to their order in the corresponding suffix of x^n. To avoid ambiguity, we will use the notation $p(a|s)$ to denote conditioning on an abstract state s, and $P(a|x_{n-j}^n)$ to denote conditioning on an arbitrary suffix of x^n, which may or may not correspond to a state. Any suffix of x^n will be called a *context* in which x_{n+1} occurs. The "basic" FSM representation is equivalent to a tree model in which all the leaves in the tree have depth m. In a *minimal* tree representation, the state $\sigma(x^n)$ for $x^n \in A_P^n$ is determined by the smallest integer $\ell(x^n)$ such that $P(\cdot|ux_{n-\ell(x^n)+1}^n)$ is independent of u, $ux_{n-\ell(x^n)+1}^n \in A_P^*$. Sets of "sibling" leaves $\{b_1b_2\dots b_{m-1}b \,|\, b \in A\}$ sharing the same CPMF in the original model can be merged into one state (leaf), represented by the parent node $b_1b_2\dots b_{m-1}$ (where (3) guarantees compatibility with the CPMF corresponding to the shorter state). The merging is repeated recursively whenever possible, seeking

[9]For example, consider a binary finite-memory process for which $P(0|u) = p$ for all strings $u \in A^+$, and $P(0|\lambda) = q$. Clearly, whenever $q \ne p$, the condition (3) is not satisfied by this process and $m = 1$, whereas $m = 0$ for $q = p$. Thus, the value of q, given by the CPMF corresponding to a transient state, determines the order of the process, a situation avoided by requiring (3).

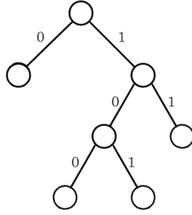

Figure 1: Binary context tree T

the shortest possible context that determines the CPMF, until any set of α sibling leaves contains at least two leaves with different associated CPMFs. A precise characterization of minimality is given in the more general setting of Section 2-B.

The minimal tree model might not be representable as an FSM with the same number of states. For example, as noted in [9], in the binary context tree of Figure 1, the state following the emission of a 1 at state 0 in T could be either 100 or 101, and more past symbols are required to make the determination than provided by the length-one context (which is nevertheless sufficient to determine the CPMF). The relation between these two classes of models will be the subject of Section 3.

In practice, the use of variable-length contexts often yields significant savings in model size compared to a full balanced tree. It is due to these savings that the theory and practice of *context tree models* based on full trees have received much attention in the literature, and efficient methods for model optimization have been developed (see, e.g., [2, 9, 3, 14]). There might be other opportunities for model size reduction, however, that are difficult to exploit using a full tree. Full tree models, for example, do not provide a mechanism for merging a proper subset of sibling leaves sharing a common CPMF into a single state.[10] We next present a more general class of tree models that could exploit some of these additional relations and provide a more economical parametrization of the process. Although this feature makes the general class interesting in itself, our main motivation in discussing it here is its use for auxiliary data structures in sections 3 and 4.

2-B Generalized context trees

Terminology and notation. For the remainder of the paper, the variables a, b, and c will always represent symbols from A, and r, s, t, u, v, w, x, y, and z will represent strings in A^*. Consider a finite, rooted, ordered, and directed tree T (see, e.g., [28, 29] for tree terminology) with the following properties:

1. Each edge is labeled with a string from A^+.

2. Each node has one incoming edge, except for the *root* of the tree, which has none. Each node has at most α outgoing edges, which must be labeled with strings starting with different symbols from A.

[10]The compression algorithm of [23] leads in some cases to such merges, although the model is not formalized.

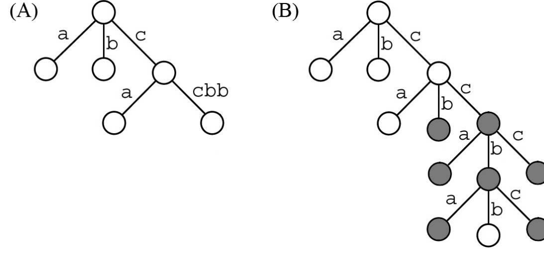

Figure 2: A GCT T over $\{\texttt{a,b,c}\}$ and the corresponding T_{full} (with added nodes in gray)

3. Each node is labeled with a finite string, obtained by concatenating the labels of the edges on the path from the root to the node. The root is labeled with λ.

For simplicity, we do not distinguish between nodes and their labels, and, for $w \in A^*$, we use the expression "w is a node of T" as shorthand for "T has a node labeled with w." Furthermore, we identify T with its set of nodes, and we write, for instance, $u \in T$ when u is a node of T. We denote the number of outgoing edges of a node u by $\deg(u)$, and if aw is the label of an edge outgoing from u, we say that this edge is *in the direction* of a. If T has an edge labeled w, going from node u to node v, we write $u \xrightarrow{w} v$, and say that v is a *child* of u, and that u is the *parent* of v, denoted $u = \textsc{par}_T(v)$. The set of children of a node u is denoted $\textsc{chld}_T(u)$. A node v is a *descendant* of u if $u \preceq v$ (u is then an *ancestor* of v). An edge of T is said to be *atomic* if it is labeled with a single-letter string; otherwise it is said to be *composite*. If $u \xrightarrow{a} ua$ is an atomic edge, then ua is an *atomic child* of u. A tree T is atomic if every edge of T is atomic. We recall that T is defined to be *full* if it is atomic and every node has either no outgoing edges (in which case it is a *leaf*), or it has α outgoing edges, one for each symbol in the alphabet. A string w is a *word* of T if it is a prefix of a node of T. The set of words of T will be denoted $\textsc{word}(T)$. Thus, by our convention of identifying the symbol T with its set of nodes, we have $T \subseteq \textsc{word}(T)$, with equality holding if and only if T is atomic.

The combinatorial structure just described has been widely used, under various guises and terminologies, as an underlying data structure for efficient string processing algorithms. The structure (or variants sharing many of its properties) has been referred to as an A^+ *tree* [16], a *PATRICIA* tree [30, 29, 31], a *compact digital tree* [31], etc. It has found numerous applications, for instance, in string storing, searching and retrieval [29, 31], pattern matching [32, 33, 16], and in the mentioned works [21, 20, 17] related to data compression, to list just a few (we cite a few references which contain extensive bibliographies; an exhaustive listing of references for the different variants and applications of digital trees would be far beyond the scope of this paper). To emphasize the application of interest in this paper, we will refer to T as a *generalized context tree* (GCT). An example of a GCT over $A = \{\texttt{a,b,c}\}$ is shown in Figure 2(A).

Source definition. We next describe how a GCT defines the state function of a recursive model which generates a string process. For a GCT T, and an arbitrary string $y \in A^*$, we define the *canonical decomposition* of y with respect to T as the triplet $C_T(y) = \langle r, u, v \rangle$ such that $r, u, v \in A^*$, r is the longest prefix of y that is a node of T, ru is the longest prefix of y that is a word of T, and $y = ruv$.

Figure 3: Canonical decomposition of $\texttt{ccbab} = \langle \texttt{c}, \texttt{cb}, \texttt{ab} \rangle$

The decomposition is illustrated in Figure 3 with an example, taken over the GCT T of Figure 2(A). Notice that v is the suffix of y that "falls off" the tree. In general, any, or all, of r, u and v may be null strings. A similar notion of *canonical reference* was defined in [16]. As we will often make separate reference to it, we will denote the first component, r, of $C_T(y)$ by $V_T(y)$.

Let \$ be a symbol such that $\$ \notin A$. Given a sequence x^n and a GCT T, we define the *tree-state function* $s_T : A^* \to A^* \cup \{\, w\$ \,|\, w \in A^* \,\}$ as follows:

$$s_T(x^n) = \begin{cases} V_T(\overline{x}^n) & \text{if } V_T(\overline{x}^n z) = V_T(\overline{x}^n) \quad \forall z \in A^*, \\ \overline{x}^n\$ & \text{otherwise.} \end{cases} \tag{4}$$

Since T is finite, the first case of (4) must hold for sufficiently large n, making $s_T(x^n)$ a node of T. For small values of n, the second case in (4) may hold. Thus, viewing s_T as a state function, its permanent states are given by all the nodes s of T for which there exist arbitrarily long sequences $y^n \in A^*$ satisfying $V_T(y^n) = s$, whereas its transient states are arbitrary words of T other than leaves, with the symbol \$ appended. By extension, we call these nodes and words, respectively, permanent and transient states of T. A transient state $u\$$ accepts only the single string $\overline{u}$, which is not long enough to "fall off" the tree or reach a leaf. We denote the set of permanent states of T by S_T, the set of transient states of T by $S_T^{\$}$, and the set of all states of T by $S_T^{\mathrm{A}} = S_T \cup S_T^{\$}$.

As an example, for the GCT shown in Figure 2(A), we have $S_T = \{\texttt{a},\texttt{b},\texttt{c},\texttt{ca},\texttt{ccbb}\}$ and $S_T^{\$} = \{\lambda\texttt{\$},\texttt{c\$},\texttt{cc\$},\texttt{ccb\$}\}$. In this example, $\texttt{c} \in S_T$ but we still have $s_T(\texttt{cc}) = \texttt{cc\$}$, namely, the second case in (4) holds. The extra symbol \$ serves to distinguish states that would otherwise correspond to the same string from A^*, e.g., $\texttt{c}$ and $\texttt{c\$}$ in the example. A natural interpretation of this symbol, which will be more explicitly adopted in Section 4, is that of a conceptual marker preceding the first actual symbol of x^n.

The following lemma summarizes the above discussion, characterizing permanent and transient states by giving formal meaning to situations in which a sequence "falls off the tree."

Lemma 1 *A string s is a permanent state of T if and only if $s \in T$ and either $\deg(s) < \alpha$ or s has a composite outgoing edge. A string $w\$$ is a transient state of T if and only if $w \in \textsc{word}(T)$ and w is not a leaf of T.*

When T is a full tree, the set of permanent states is identical to the set of leaves. For the full binary GCT of Figure 1, for example, we have $S_T = \{0, 100, 101, 11\}$ and $S_T^{\$} = \{\lambda\$, 1\$, 10\$\}$.

We denote with $\langle T, p\rangle$ the recursive model defined by the state set S_T^{A}, the state function $s_T(\cdot)$ of (4), and an associated set of CPMFs $\{p(\cdot|s)\}$, $s \in S_T^{\mathrm{A}}$. The probability assignment (1) generated by $\langle T, p\rangle$ clearly has finite-memory, with order m upper-bounded by the length of the longest word of T. In order to satisfy also (3), it suffices to require that if $s\$$ is a transient state such that all permanent states of the form $V_T(su)$, $u \in A^*$, share the same CPMF, then this CPMF is also associated with $s\$$.
Remark. Our definitions are quite general in letting arbitrary words define transient states of the GCT, and allowing arbitrary CPMFs to be associated with these states, as long as (3) is satisfied. A popular convention is to use for a transient state the CPMF associated with the permanent state that would be selected had the sequence been preceded by as many copies of a fixed symbol as needed [2]. In the context of source coding, another reasonable convention is to assume that transient states are associated with uniform distributions. We will specify a particular CPMF choice for transient states only when required for the results. For example, the results of Section 5 require this choice to be consistent with the stationary distribution of the Markov chain associated with the CPMFs of permanent states.

Relation to full-tree models. The conventional full-tree models are a special case of GCTs. For any model $\langle T, p\rangle$, the GCT T can be completed to a full tree T_{full}, for which there exists a probability assignment p' such that $\langle T, p\rangle$ and $\langle T_{\mathrm{full}}, p'\rangle$ generate the same process, as follows. Let s be a permanent state of T, with associated CPMF $p(\cdot|s)$. Then,

a. if s is a leaf of T, then s is a state (leaf) of $S_{T_{\mathrm{full}}}$ and $p'(\cdot|s) = p(\cdot|s)$;

b. otherwise, for every $a \in A$ such that s does not have an edge in the direction of a, sa is a state of $S_{T_{\mathrm{full}}}$ and $p'(\cdot|sa) = p(\cdot|s)$;

c. for every composite edge aw emanating from s, with $w = w_1^\ell$, $\ell \geq 1$, all the strings $saw_1^i c_i$, $0 \leq i < \ell$, $c_i \in A \setminus \{w_{i+1}\}$, are states of $S_{T_{\mathrm{full}}}$, sharing the CPMF $p(\cdot|s)$.

d. $S_{T_{\mathrm{full}}}^{\$} = S_T^{\$}$, and for any $w\$ \in S_T^{\$}$ we have $p'(\cdot|w\$) = p(\cdot|w\$)$.

It is possible, therefore, for S_T to be significantly smaller than $S_{T_{\mathrm{full}}}$, providing a more economical parametrization of the process. In other words, a minimal model in the full-tree sub-class may still be reducible in the GCT class. Part (B) of Figure 2 shows the underlying full tree T_{full} corresponding to the GCT in Part (A) of the same figure. In the example, we have $|S_T| = 5$ and $|S_{T_{\mathrm{full}}}| = 9$. Later on in this sub-section, and in Appendix A, we characterize minimal representations in the GCT class. However, the current state of the art in modeling algorithms does not allow us to efficiently optimize code length in this class. Thus, we cannot take advantage of the additional flexibility. The GCT extension will be used in our case as an algorithmic tool for dealing with suffix trees which may not be full, in order to achieve the complexity results of Section 4. The code length optimized, however, will still correspond to the sub-class of full-tree models.

Normal GCTs. We next present a partition of the set of GCTs into equivalence classes. This partition simplifies the derivation of further results. Given a GCT T, we say that a node $v \in T$ is a *pseudo-leaf* if $\deg(v) \leq 1$ (the case $\deg(v) = 0$ corresponds to a leaf). We say that v is a *phantom node* of T if $v \notin T$, and $v = ua$, where $a \in A$, $u \in T$, and for every $b \in A \setminus \{a\}$, $ub \in T$. By Lemma 1, $u \in S_T$.

If we add ua as a node to T (by either adding or splitting an edge), it becomes a pseudo-leaf and a permanent state accepting the same set of strings previously accepted by u which, again by Lemma 1, ceases to be a permanent state. Also, since the set of words of the GCT which are not leaves remains unchanged, so does the set of transient states. Thus, the sets of states of the two GCTs are in one-to-one correspondence. Moreover, for a GCT model $\langle T, p\rangle$, if we also associate with the added node ua the CPMF $p(\cdot|u)$, then the new GCT model generates the same process as the original one. Thus, a GCT T with a phantom node ua is indistinguishable, from the point of view of the properties of interest to us, from $T \cup \{ua\}$.

We call the operation of replacing a phantom node of a GCT with the actual node a *normalization* step, and we call a GCT without phantom nodes *normal*. For $\alpha = 2$, normalization might be a two-step process, in that replacing a phantom node with the actual node by splitting a composite edge labeled with a string *of length two*, creates another phantom node, which in turn needs to be replaced by adding a leaf to the new node. Clearly, this situation does not occur for $\alpha > 2$. One can also take an "unnormalization" step by eliminating a pseudo-leaf from a full set of sibling nodes. Again, in the binary case, this step could create another "unnormalizable" pseudo-leaf. Notice that a full tree is always normal.

The normalization/unnormalization operations define a partition of the set of all α-ary GCTs into classes, where two GCTs belong to the same class if and only if one can be obtained from the other through a finite sequence of normalization/unnormalization operations. Let $\mathbf{N}(T)$ denote the class of T in this partition. Clearly, $\mathbf{N}(T)$ contains one and only one normal GCT T_N, which we call the *normalized presentation* of T. The GCT T_N can be obtained from T by replacing each phantom node with an actual node (and, in the binary case, possibly adding leaves as noted, so that no phantom nodes are left). Also, note that $T_N = \bigcup_{T' \in \mathbf{N}(T)} T'$.

Minimal GCT models. A GCT model $\langle T, p\rangle$ is said to be *minimal* if no other GCT model $\langle T', p'\rangle$ generates the same process and has a smaller number of permanent states.[11] To characterize minimality, we start with the conventional sub-class of full-tree models, for which the characterization is simple and well known (see, e.g., [3]). For completeness, we derive this characterization in Lemma 2 below. We say that a GCT T' is an *extension* of a GCT T if it contains all the nodes of T.

Lemma 2 *A full-tree model $\langle T, p\rangle$ with $A^* = A_P^*$ is minimal if and only if there is no set of α sibling leaves of T sharing the same CPMF. Moreover, if $\langle T, p\rangle$ is minimal, and $\langle T', p'\rangle$ generates the same process, where T' is also full, then T' is an extension of T.*

Proof. The necessity of the minimality condition is straightforward, since sets of sibling leaves with identical CPMFs can always be merged, reducing the number of states (constraint (3) guarantees that transient CPMFs do not impede the merging). Assume the condition holds, and $\langle T', p'\rangle$ generates the same process as $\langle T, p\rangle$, with T' full. Assume u is a node in $T \setminus T'$. Then, there is a leaf $u' \in T'$ such that $u' \prec u$, and there is a full set of sibling leaves of T that descend from u'. But, since $u' \in S_{T'}$, $A_P^* = A^*$, and both tree models generate the same process, these leaves of T must be associated with the same CPMF that is associated with u' in T', contradicting the assumed condition. Thus, we must

[11]While we emphasize the permanent states because they determine the lasting statistics of the source, it can be shown that a minimal GCT model is also minimal in its number of transient states.

have $T \subseteq T'$, which also establishes the minimality of T. □

The situation is far more complex for the GCT class. Since the resulting characterization is not needed for the results in the sections to follow, its discussion and proof are deferred to Appendix A.

3 FSM closures of generalized context trees

As discussed in Section 2, FSMs and GCTs are combinatorial mechanisms used for process generation, providing the state function σ of a recursive model. For a GCT T, σ is given by the tree-state function s_T and $s_0 = \lambda\$$, whereas for FSMs σ is recursively defined by the next-state function, starting from an initial state s_0. The class of FSM-generated processes properly includes finite-memory processes (see, e.g. [26]). However, as shown by the example in Figure 1, a minimal tree representation of a finite-memory process might have fewer states than an FSM representation of the same process. In this section, we study the relation between these two process-generating mechanisms, we define the FSM closure of a GCT, and we present an efficient algorithm for constructing it.

3-A Refinements

We now study structural relations between recursive models that generate the same process, and develop tools that will prove useful in investigating the FSM closure of a GCT. Let σ and σ' be state functions taking values in state sets S and S', respectively. We say that σ' is a refinement of σ if there exists a *refinement function* $g : S' \to S$ such that for all sequences x^n, if $\sigma'(x^n) = s'$ and $\sigma(x^n) = s$, then $g(s') = s$. This notion of refinement was presented in [34] for FSMs, and is used also in [35]. We will loosely identify state functions with the mechanisms defining them and say, e.g., that an FSM $\mathcal{F}$ is a refinement of a GCT T.

Lemma 3 *Let σ and σ' denote state functions taking values in state sets S and S', respectively.*

(i) *If σ' is a refinement of σ with refinement function g, then for any set of CPMFs $\{p(\cdot|s)\}_{s\in S}$ there exists another set $\{p'(\cdot|s')\}_{s'\in S'}$ such that $\langle\sigma, p\rangle$ and $\langle\sigma', p'\rangle$ generate the same process, and $p'(\cdot|s') = p(\cdot|g(s'))$ (as functions).*

(ii) *Conversely, if for a set of distinct CPMFs $\{p(\cdot|s)\}_{s\in S}$ there exists another set $\{p'(\cdot|s')\}_{s'\in S'}$ such that $\langle\sigma, p\rangle$ and $\langle\sigma', p'\rangle$ generate the same process P, and $A^*_P = A^*$, then σ' is a refinement of σ.*

Proof. First, notice that for two recursive models $\langle\sigma, p\rangle$ and $\langle\sigma', p'\rangle$ that generate the same process P, if $u \in A^*_P$ selects $s = \sigma(u)$ and $s' = \sigma'(u)$, then, for any $c \in A$, we have

$$p'(c|s') = \frac{P(uc)}{P(u)} = p(c|s) \tag{5}$$

where both equalities follow from (1) (as $u \in A^*_P$). Now, To prove the first part of the lemma, the set of CPMFs $\{p'(\cdot|s')\}_{s'\in S'}$ defined by

$$p'(\cdot|s') = p(\cdot|g(s')) \ \ \forall s' \in S' \tag{6}$$

clearly satisfies that $\langle\sigma, p\rangle$ and $\langle\sigma', p'\rangle$ generate the same process. Moreover, by (5) and the definition of refinement, any set $\{p'(\cdot|s')\}_{s'\in S'}$ such that $\langle\sigma, p\rangle$ and $\langle\sigma', p'\rangle$ generate the same process must satisfy (6). Conversely, assume that for a set of distinct CPMFs $\{p(\cdot|s)\}_{s\in S}$ there exists another set $\{p'(\cdot|s')\}_{s'\in S'}$ such that $\langle\sigma, p\rangle$ and $\langle\sigma', p'\rangle$ generate the same process, and $A_P^* = A^*$. If σ' is not a refinement of σ, there exist two sequences u and v that select the same state s' of S' but two different states s_u and s_v of S, respectively. By (5), since $A_P^* = A^*$ we have $p(\cdot|s_u) = p'(\cdot|s') = p(\cdot|s_v)$. Thus, the identity $p(\cdot|s_u) = p(\cdot|s_v)$ contradicts the assumption that the CPMFs in $\{p\}$ are distinct. □

Lemma 3 implies, a fortiori, that σ' is a refinement of σ if and only if any process generated by a model of the form $\langle\sigma, p\rangle$ can also be generated by a model of the form $\langle\sigma', p'\rangle$. While our original definition of a refinement (which does not involve any process) targets the application of FSM closures in Section 4, this alternative characterization is convenient for some of the arguments in Section 5.

The following lemma characterizes the FSM refinements (if any) of a given state function σ.

Lemma 4 *Let the state function σ admit an FSM refinement, and let $\mathcal{F}$ denote the FSM refinement of σ with the least number of states. Then, all the FSM refinements of σ are also refinements of $\mathcal{F}$.*

Proof. Let $\mathcal{F} = (S, f, s_0)$, let g denote the corresponding refinement function with respect to σ, and let $\mathcal{F}' = (S', f', s_0')$ denote another FSM refinement of σ, with refinement function g'. We will extend our notation by denoting $f(s, z)$ the state reached by $\mathcal{F}$ after emission of $z \in A^*$, starting at $s \in S$ (the same abuse of notation applies to f'). Suppose $\mathcal{F}'$ is not a refinement of $\mathcal{F}$. Then, there exist two sequences, z_1 and z_2, which select the same state $w \in S'$, but two different states u and v, respectively, in S. For an arbitrary sequence $z \in A^*$, z_1z and z_2z select the same state for $\mathcal{F}'$, and consequently also for σ. Therefore, we have

$$g(f(u, z)) = g'(f'(w, z)) = g(f(v, z))\,. \tag{7}$$

Clearly, (7) implies that if we delete u from $\mathcal{F}$ and redirect all its incoming edges to v, the resulting FSM will still be a refinement of σ. This FSM has fewer states than $\mathcal{F}$, a contradiction. □

We now focus on refinement relations between GCTs, using the partition, defined in Section 2, of the set of all α-ary GCTs into equivalence classes of GCTs sharing a common normalized presentation. Lemma 5 below is an obvious consequence of our discussion on normalization.

Lemma 5 *If $\mathbf{N}(T^*) = \mathbf{N}(T)$ then there exists a one-to-one refinement mapping between T^* and T.*

Next, we relate the notion of refinement more directly to the combinatorial structure of a GCT.

Lemma 6 *Let T and T' be GCTs. T' is a refinement of T if and only if T_N' is an extension of T, where T_N' is the normalized presentation of T'.*

Proof. Assume first that $T \subseteq T_N'$. Consider a sequence of transformations in which we start with T, and we add one node of $T_N' \setminus T$ at a time, until we obtain T_N'. Let T^* denote a generic tree obtained at

an intermediate step of this process. Since $T' \in \mathbf{N}(T'_N)$, by transitivity of the refinement and Lemma 5, it suffices to prove that every addition step in this process produces a refinement T^*_r of T^*.

The addition of a node v can be the result of either adding an outgoing edge to a node u of T^* in a direction in which u did not have an edge, or splitting an outgoing composite edge of u, inserting v. Clearly, in either case, $u \in S_{T^*}$, $\{v\} = S_{T^*_r} \setminus S_{T^*}$, and $S_{T^*} \setminus S_{T^*_r}$ is either $\{u\}$ or empty. Thus, a refinement function g^* is defined such that $g^*(v) = u$ and $g^*(z) = z$ for all $z \in S_{T^*_r} \setminus \{v\}$. As for the transient states, $S^{\$}_{T^*_r} = S^{\$}_{T^*}$ (thus defining an identity mapping), unless u is a leaf of T^*, in which case $S^{\$}_{T^*_r} = S^{\$}_{T^*} \cup \{u\$\}$. Clearly, in the latter case, $g^*(u\$) = g^*(u)$. Hence, T'_N is a refinement of T.

Assume now that T' is a refinement of T. Then, by transitivity of the refinement and Lemma 5, there exists a refinement function $g : T'_N \to T$. If $T \not\subseteq T'_N$, there exists a node $w \in T \setminus T'_N$. Clearly, $w \neq \lambda$, so we assume $w = ua$ for some $u \in A^*$, $a \in A$. Since $ua \notin T'_N$, for some $y \in A^*$, we have $s_{T'_N}(ya\overline{u}) = u' \preceq u$, and $s_T(ya\overline{u}) = g(u') \succeq ua$. Write $ua = u'bv$, $b \in A$. We claim that for all $d \in A \setminus \{b\}$, we must have $u'd \in T'_N$. Otherwise, if $u'd \notin T'_N$, by Lemma 1, we would have $s_{T'_N}(zd\overline{u}') = u'$ for some z, and $s_T(zd\overline{u}') \not\succeq ua$, a contradiction to our previous determination of $g(u')$. Now, since T'_N is normal, we must also have $u'b \in T'_N$, for otherwise $u'b$ would be a phantom node of T'_N. Thus, we have a contradiction to the fact that $V_{T'_N}(u'bv) = u'$. Therefore, $T \subseteq T'_N$. $\square$

Remarks

(a) It follows from Lemma 6 that the notions of refinement and extension coincide for normal trees, and, thus, for all full trees. Lemma 6 also implies that the sufficient condition given in Lemma 5 for the existence of a one-to-one refinement mapping between two GCTs is also necessary.

(b) The notions of refinement and minimality were related in [35] for FSM models. An FSM model is minimal if no other FSM model can generate the same process with fewer states. It is shown in [35, Lemma 1] that if $\langle \mathcal{F}, p \rangle$ and $\langle \mathcal{F}', p' \rangle$ generate the same process, and $\mathcal{F}$ is minimal, then $\mathcal{F}'$ is a refinement of $\mathcal{F}$. While an analogous result holds for full-tree models (see Lemma 2), and for ternary normal GCT models (see Theorem A.1 in Appendix A), it is interesting to notice that this property does not hold, in general, for GCT models with $\alpha \neq 3$, as shown by the examples, given in Appendix A, of multiple minimal GCT models of the same process.[12]

3-B Definition and properties of FSM closures

We say that an FSM $\mathcal{F}$ is an *FSM closure* of T if it is a refinement of T with minimal number of permanent states. As in the definition of a minimal GCT model, we adopt the number of permanent states in $\mathcal{F}$ as the most relevant measure of minimality. However, Lemma 7 below shows that, in fact, there exists an FSM closure of T that is also minimal in the stronger sense of having a minimal (total) number of states.

[12] These multiple minimal GCT models, however, may be proper refinements of a minimal FSM model (not derived from a GCT). This is the case in the example given in Appendix A for $\alpha = 4$, where it is easy to see that the process admits a minimal FSM model with two recurrent states (and one transient state).

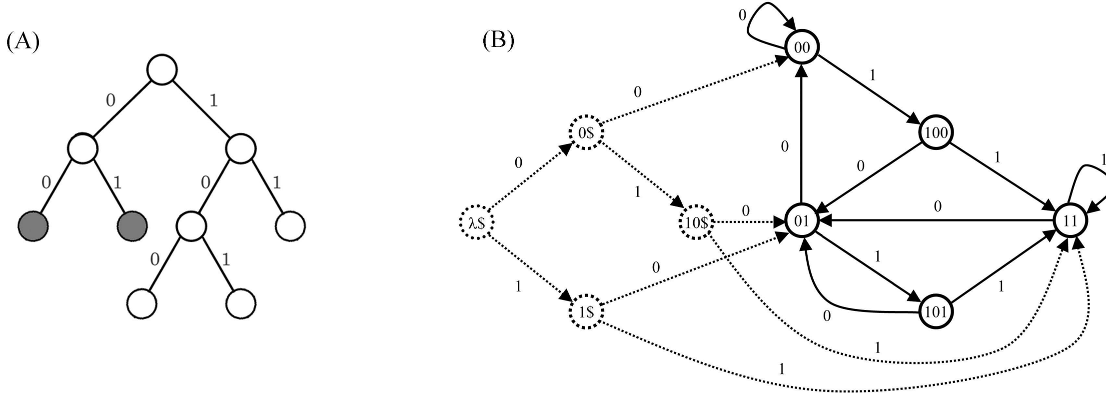

Figure 4: FSM closure T_F of binary GCT T and corresponding finite state machine

Lemma 7 *Let $\mathcal{F}$ be an FSM refinement of a GCT T having a minimal (total) number of states. Then, any FSM refinement of T is a refinement of $\mathcal{F}$ and $\mathcal{F}$ is, a fortiori, an FSM closure of T.*

Proof. The fact that any FSM refinement $\mathcal{F}'$ of T is a refinement of $\mathcal{F}$ is a direct result of Lemma 4. Clearly, if g denotes the corresponding refinement function and s is a permanent state of $\mathcal{F}$, there exists at least one permanent state s' of $\mathcal{F}'$ such that $g(s') = s$, for otherwise a transient state of $\mathcal{F}'$ would accept arbitrarily long strings. Thus, $\mathcal{F}'$ has at least as many permanent states as $\mathcal{F}$ and, consequently, $\mathcal{F}$ is an FSM closure of T. $\square$

We say that a GCT T has the *FSM property* if it defines a next-state function $f : S_T^{\mathrm{A}} \times A \to S_T^{\mathrm{A}}$ such that, for any sequence x^{n+1}, we have

$$s_T(x^{n+1}) = f(s_T(x^n), x_{n+1}).$$

For brevity, when T has the FSM property we say that "T is FSM," and we do not distinguish between T and the corresponding FSM. Clearly, if T is FSM then it is also an FSM closure of T. The FSM property facilitates the implementation of GCT models, due to the recursive form of the next-state function. However, as discussed in Section 2 and exemplified in [9] and Figure 1, a GCT may not be FSM. In such cases, its FSM closure is instrumental for efficient implementation.

Figure 4(A) shows a GCT T_F with the FSM property which is an FSM closure of the GCT T of Figure 1. New nodes added to T are shaded. Figure 4(B) shows the finite state machine associated with T_F. Transient states and their transitions are shown with dashed lines. While the FSM closure of T shown in this example is itself a GCT, it is conceivable that, in general, an FSM refinement which is not constrained to having an underlying GCT structure (namely, one that does not correspond to a GCT with the FSM property), might have fewer permanent states than any FSM refinement that is also a GCT. Next, we show this not to be the case, and we characterize an FSM closure of a GCT. We start with a sufficient condition for a GCT to have the FSM property.

Lemma 8 *Let T be a GCT. If for every permanent state $s \in S_T$, the suffix* $\text{tail}(s)$ *is a node of T, then T is FSM and the next-state function f satisfies, for all $a \in A$, $f(s,a) = V_T(as)$.*

Proof. We show that a next-state function f can be defined for T. Let $s \in S_T$, and let $w\overline{s}$ be a string accepted by s, $w \in A^*$. For any $a \in A$ we have $as\overline{w} \notin \text{WORD}(T)$, for otherwise $as\overline{w}$ is a prefix of a permanent state and, by the lemma assumption, we would have $s\overline{w} \in \text{WORD}(T)$ implying that $w\overline{s}$ selects a transient state. Thus, $w\overline{s}a$ selects a permanent state $r = V_T(as\overline{w})$. Clearly, $r \preceq as$, for otherwise $s \prec \text{tail}(r)$ and, by the lemma assumption, $\text{tail}(r) \in T$, implying that $w\overline{s}$ would not have selected s. Therefore, $r = V_T(as)$, and we can define the state transition $f(s,a) = V_T(as)$.

For a transient state $z = u\$$ of T, if $au \in \text{WORD}(T)$ then we define $f(z,a) = az$. Otherwise, au selects the permanent state $V_T(au)$, and we can define $f(z,a) = V_T(au)$. The next-state function is now defined for all states $s \in S_T^{\text{A}}$, and, hence, T is FSM. □

Consider the GCT T_{suf} obtained from a GCT T by adding, as nodes, all the suffixes of nodes of T. Notice that the addition of a node may cause a composite edge to split. Thus, T_{suf} might contain nodes that are added to satisfy structural constraints of the tree, rather than directly as suffixes of nodes of T. For example, if w is a node of T with an outgoing edge uv, and the construction calls for adding the node wu, then the edge $w \xrightarrow{uv} wuv$ is split as $w \xrightarrow{u} wu \xrightarrow{v} wuv$. The GCTs in Figures 1 and 4(A) satisfy $T_F = T_{\text{suf}}$. The suffix 00 of state 100 of T is not a node of T, and therefore T does not satisfy the sufficient condition of Lemma 8. We can now state the main result of this section.

Theorem 1 *Let T be a GCT. Then, T_{suf} is an FSM closure of T.*

Proof. Since T_{suf} is an extension of T, by Lemma 6, it is also a refinement of T. Moreover, from the definition of T_{suf} and from Lemma 8, it follows immediately that T_{suf} is FSM.

To prove minimality, let $\mathcal{F} = (S, f, s_0)$ denote another FSM refinement of T, having a minimal (total) number of states. By Lemma 7, T_{suf} is a refinement of $\mathcal{F}$, and let g denote the corresponding refinement function. We will show that the existence of $u, v \in S_{T_{\text{suf}}}$, $u \neq v$, such that $g(u) = g(v) = s \in S$, leads to a contradiction, proving that the restriction of g to $S_{T_{\text{suf}}}$ is a one-to-one mapping between $S_{T_{\text{suf}}}$ and the set of permanent states of $\mathcal{F}$, and, consequently, T_{suf} is an FSM closure of T. To this end, we first observe that since $\mathcal{F}$ is a refinement of T, by transitivity, u and v correspond to the same (permanent) state w of T in the refinement $\varphi : S_{T_{\text{suf}}}^{\text{A}} \to S_T^{\text{A}}$. Since T_{suf} is an extension of T, w must be a common prefix of u and v and, without loss of generality, we can assume that $v \not\preceq u$, so that $w \prec v$ and $w \preceq u$.

Consider strings $x, y \in A^*$ such that $S_{T_{\text{suf}}}(x\overline{u}) = u$ and $S_{T_{\text{suf}}}(y\overline{v}) = v$. Both $x\overline{u}$ and $y\overline{v}$ bring $\mathcal{F}$ to the same state s, and we must also have $S_T(x\overline{u}) = S_T(y\overline{v}) = w \prec v$, so that $v \notin T$. Therefore, since $v \in T_{\text{suf}}$, by the definition of T_{suf}, there exists $t \in A^+$ such that $tv \in T$. Next, consider emission of $\overline{t}$ after $x\overline{u}$ and after $y\overline{v}$. In both cases, $\mathcal{F}$ evolves from state s to some $s' \in S$, and let $w' = \varphi(s')$ be the corresponding (permanent) state of T. Since $tv \in T$ and $w' = V_T(tv\overline{y})$, we must have $tv \preceq w'$. On the other hand, $w' = V_T(tu\overline{x}) \preceq tu$, for otherwise we would have $tur \in T$, with $r \in A^+$ and $r \preceq \overline{x}$, implying that $ur \in T_{\text{suf}}$, in contradiction with $S_{T_{\text{suf}}}(x\overline{u}) = u$. Thus, we have $tv \preceq w' \preceq tu$, in contradiction with $v \not\preceq u$. □

Theorem 2 below fully characterizes GCTs having the FSM property.

Theorem 2 *A GCT T with normal presentation T_N is FSM if and only if every suffix of a node of T_N is a node of T_N.*

Proof. By Lemma 5, we can assume without loss of generality that T is normal. If T is FSM, it must have as many permanent states as its FSM closure $T_{\rm suf}$. Since T has no phantom nodes, any node added in order to extend it to $T_{\rm suf}$ will increase the number of permanent states. Thus, we must have $T_{\rm suf} = T$, and the claim follows from the definition of $T_{\rm suf}$. Conversely, if every suffix of a node of T is a node of T, T is FSM by Lemma 8. $\square$

Remarks

(a) It can be shown that the FSM derived from $T_{\rm suf}$ by deleting all transient states $u\$$ such that $u \notin \textsc{word}(T) \cup T_{\rm suf}$ and redirecting the corresponding incoming edges to $V_T(u)\$$ is an FSM closure of T with a minimal (total) number of states. We omit the proof.

(b) By Lemma 4, the permanent state sets of any two FSM closures of a GCT T are in one-to-one correspondence, which extends to the state sequences followed by any string. Thus, all FSM closures are essentially equivalent, differing possibly only in the transient states, which are of little interest to us. Therefore, we will henceforth refer to $T_{\rm suf}$ as *the* FSM closure of T.

(c) If T is atomic, then T is FSM if and only if every substring of a node of T_N is a node of T_N, since in such a tree every prefix of a node is a node.

(d) It is readily verified that if T is full, so is $T_{\rm suf}$.

(e) By Theorem 2, for any normal GCT T_F that is an FSM refinement of T we have $T \subseteq T_{\rm suf} \subseteq T_F$. Thus, if $T_{\rm suf}$ is normal, it is the *only* normal GCT which is an FSM closure of T. This property holds in particular for full trees.

3-C The size of the FSM closure

We now investigate the size of $T_{\rm suf}$, and, in particular, how it compares with that of T. We will focus our attention on full trees, mainly because those will be the relevant models in our application of these results in Sections 4 and 5. Also, we are interested in finding functional relations and bounds between the sizes of T and $T_{\rm suf}$. In the case of full trees, these relations will become clear, regardless of the precise definition of "size." The situation is not as well-defined in the case of GCTs with arbitrary trees. For example, consider the GCT $T = \{\lambda, w^k\}$. This GCT has two nodes (and permanent states). The number of nodes (and states) of $T_{\rm suf}$, on the other hand, is a function of k, which is arbitrary. The sizes could be measured in terms of the sum of the lengths of the strings that label the edges, and a more meaningful functional relation would be obtained. However, this measure does not seem relevant to the issues of interest in our applications in Sections 4 and 5.

Let K denote the number of leaves (permanent states) of a full α-ary tree T, and $|T|$ the total number of its nodes. As is well known (see, e.g., [28, p. 595]), $K = ((\alpha-1)|T|+1)/\alpha$, or, equivalently,

$$|T| = \frac{\alpha K - 1}{\alpha - 1}\,. \tag{8}$$

Lemma 9 *Let T be a full α-ary tree with K leaves, and $T_{\rm suf}$ its FSM closure. Then,*

$$|T_{\rm suf}| \le \frac{\alpha}{2(\alpha-1)^2}K^2 + \frac{\alpha(\alpha-3)}{2(\alpha-1)^2}K + \frac{(\alpha-1)^2+1}{2(\alpha-1)^2}. \tag{9}$$

Proof. We prove the claim by induction on K. If $K=1$, the tree consists just of the root, $|T_{\rm suf}|=1$, and the validity of (9) (with equality), is verified by straightforward algebraic manipulations. Assume now the claim holds for $1 \le K' < K$, let T_a, $a \in A$, denote the subtrees of T rooted at the children of the root of T, and K_a the number of leaves of T_a. By definition, the nodes of $T_{\rm suf}$ are all the suffixes of nodes of T. A suffix of a node of T is either a suffix of a node of one of the T_a's, or of the form au_a, where u_a is a node of the subtree T_a in the direction of a from T's root (u_a is read as a word of T_a). Therefore, we can write

$$|T_{\rm suf}| \le \sum_{a\in A} |(T_a)_{\rm suf}| - (\alpha-1) + \left(\frac{\alpha K - 1}{\alpha-1} - 1\right), \tag{10}$$

where the first term in (10) bounds the number of suffixes of nodes of the subtrees T_a, the second term compensates for over-counting the string λ, which is guaranteed to be in the suffix set of all the subtrees, and the last term is, by (8), the number of nodes in the subtrees. Let $\beta = \frac{\alpha}{2(\alpha-1)^2}$, $\gamma = \frac{\alpha(\alpha-3)}{2(\alpha-1)^2}$, and $\delta = \frac{(\alpha-1)^2+1}{2(\alpha-1)^2}$. Then, the expression at the right hand side of (9) can be written as $f(K) = \beta K^2 + \gamma K + \delta$. By the induction hypothesis, we have $|(T_a)_{\rm suf}| \le f(K_a)$, and, thus

$$\begin{aligned}
|T_{\rm suf}| &\le \sum_{a\in A} f(K_a) - (\alpha-1) + \frac{\alpha K - 1}{\alpha-1} - 1 \\
&= \beta \sum_{a\in A} K_a^2 + \gamma K + \alpha\delta - (\alpha-1) + \frac{\alpha}{\alpha-1}(K-1) \\
&= \beta \sum_{a\in A} K_a^2 + \left(\gamma + \frac{\alpha}{\alpha-1}\right) K + \left(\alpha\delta - \frac{\alpha^2}{\alpha-1} + 1\right).
\end{aligned} \tag{11}$$

The first term in the last line of (11) is proportional to a sum of squares of positive integers, constrained by $\sum_a K_a = K$. Such a sum attains its maximum value when one of the numbers is as large as possible, while the others are kept at their minimum value, i.e., say, $K_b = K - \alpha + 1$ for some $b \in A$, and $K_c = 1,\ \ c \in A \setminus \{b\}$. Thus, we have $\sum_{a\in A} K_a^2 \le (K - (\alpha-1))^2 + \alpha - 1$. Substituting in (11), and rearranging terms, we obtain the bound

$$\begin{aligned}
|T_{\rm suf}| \le \beta K^2 &+ \left(\gamma + \frac{\alpha}{\alpha-1} - 2(\alpha-1)\beta\right) K \\
&+ \left(\alpha(\alpha-1)\beta + \alpha\delta - \frac{\alpha^2}{\alpha-1} + 1\right).
\end{aligned} \tag{12}$$

Substitution of the definitions of β, γ, and δ in (12) yields (9). □

Lemma 9 gives an upper bound on the size of T_{suf}, which is quadratic in the size of T. The proof of the lemma also hints at what the "worst case" structure for T is (in the sense of inflating T_{suf} the most): take a full α-ary tree in which, at each level, all the nodes except one are leaves (i.e., a tree as deep as possible). However, the starting point in the proof of the lemma is the "union bound" type inequality (10), which can be relatively tight only if the various suffix sets whose cardinalities are being added up do not have significant intersections. We next show that a tree with this property can be explicitly constructed, attaining the upper bound of Lemma 9 up to second order terms.

Lemma 10 *For every positive integer K such that $K \equiv 1 \mod (\alpha-1)$, there exists a full α-ary tree $T_{\mathbf{b}}$ with K leaves such that*

$$|(T_{\mathbf{b}})_{\text{suf}}| = \frac{\alpha}{2(\alpha-1)^2}K^2 - O(K\log K). \tag{13}$$

Proof. The condition $K \equiv 1 \mod (\alpha-1)$ is necessary, by (8), for K to be the number of leaves of a full α-ary tree. An α-ary *de Bruijn* sequence [36] $\mathbf{b} = b_0^{\alpha^k-1}$, of order k and length α^k over A, has the following property: the sliding windows $b_i b_{i+1} \dots b_{i+k-1}$, where indices are taken modulo α^k, exhaust all distinct k-tuples over A. De Bruijn sequences exist for all alphabet sizes α and orders k, and have been extensively studied (see, e.g., [37, 38]). Consider a de Bruijn sequence $\mathbf{b}$ of order $k = \lceil \log_\alpha \frac{K-1}{\alpha-1} \rceil$. We construct a tree $T_{\mathbf{b}}$ as follows: starting at the root, construct children for all $a \in A$. All these nodes will be leaves of $T_{\mathbf{b}}$, except the child corresponding to $a = b_0$, which will be the root of a subtree where the same construction is repeated with the sequence $b_1^{\alpha^k-1}$. The construction continues for the prefix of length $L-1 = (K-1)/(\alpha-1) - 1$ of $\mathbf{b}$, until the tree has $(\alpha-1)(L-1)+\alpha = K$ leaves. By construction, the prefix $\mathbf{b}'$ of length L of $\mathbf{b}$ is one of α longest words in $T_{\mathbf{b}}$. By the properties of the de Bruijn sequence, every suffix of length k or more of $\mathbf{b}'$ starts with a unique k-tuple of symbols. Since the construction of $(T_{\mathbf{b}})_{\text{suf}}$ includes adding such suffixes of nodes of $T_{\mathbf{b}}$ as paths from the root, the added paths will be disjoint after taking $k-1$ steps from the root. Since $T_{\mathbf{b}}$ is full, so is $(T_{\mathbf{b}})_{\text{suf}}$. Therefore, all the nodes needed to complete the added paths to a full tree are also in $(T_{\mathbf{b}})_{\text{suf}}$. Thus, a suffix of length $\ell \geq k$ contributes at least a set of $(\ell-k)\alpha$ distinct nodes to $(T_{\mathbf{b}})_{\text{suf}}$, and, hence, recalling the definitions of L and k, we have

$$|(T_{\mathbf{b}})_{\text{suf}}| \geq \alpha \sum_{\ell=k}^{L} (\ell-k) = \alpha \frac{(L-k)(L-k+1)}{2} = \frac{\alpha}{2(\alpha-1)^2}K^2 - O(K\log K).$$

□

We summarize the results of lemmas 9 and 10 in the following theorem.

Theorem 3 *The largest FSM closure of any full α-ary tree T with K leaves has size*

$$|T_{\text{suf}}| = \frac{\alpha}{2(\alpha-1)^2}K^2 + o(K^2).$$

3-D A linear-time algorithm for constructing FSM closures

We present an algorithm that constructs the FSM closure T_{suf} of an arbitrary GCT T, together with the associated next-state function. We will prove that the algorithm runs in time that is linear in the sum of the lengths of the strings that label edges of T and in the total number of nodes in T_{suf}.

The algorithm starts with a representation of T, and adds the necessary nodes and edges to construct T_{suf}. At any time during the computation, we denote by T' the intermediate GCT in existence at the time. Thus, T' evolves from T to T_{suf}. When referring to canonical decompositions $C_{T'}$, we mean the decomposition with respect to the instantaneous state of T' at the time of the reference.

The algorithm is presented in the form of a main routine `MakeFSM`, and three subroutines, whose functions are broadly described as follows:

- `Verify(`w`)`: Receives a node w of T' as input, and verifies that the suffix $\text{tail}(w)$ is in T', adding it if necessary together with the FSM transition $f(\text{tail}(w), \text{head}(w)) = w$. The entire (evolving) tree is traversed and verified through recursive calls to this subroutine. Clearly, the condition verified by `Verify` is necessary and sufficient (if applied recursively to all the nodes) for the constructed tree to be T_{suf}.

- `Insert(`r`,`u`,`v`)`: Receives a node r of T', and strings u, v. Inserts, if necessary, new nodes ru and ruv, doing necessary edge splits and additions.

- `PropagateTransitions(`F`,`w`)`: For a function $F : A \to T_{\text{suf}}$ adds to the description of the FSM associated with T_{suf} a set of state transitions of the form $f(w, a) = F(a)$, originating from w, for all $a \in A$ such that $f(w, a)$ was not defined by `Verify`.

The routines maintain the following data arrays:

- `Tail[`w`]`: A pointer from the node in the tree containing w to the node containing $\text{tail}(w)$, which allows the algorithm to jump from w to its suffix in constant time. These *suffix links* [16] are essential to the efficient implementation of the algorithm.

- `Traversed[`w`,`a`]`: A flag indicating whether an attempt was made to traverse an edge starting from node w in the direction of a. Initially set to *false* for all $a \in A$ for nodes $w \in T$ as well as for new nodes as they are created.

- `Transitions[`w`]`: A function mapping A into $T' \cup \{\perp\}$, where $\perp$ denotes an undefined state. The function lists the FSM transitions from state w. The list of transitions is initially set to $\perp$ for all $a \in A$.

- `Origin[`w`]`: The original node in T that w descends from. Initially, `Origin(`w`)`$=w$. This array connects the states of the constructed FSM closure to the original states of T and their associated CPMFs, which the FSM states must inherit.

- `Children[`w`]`: The list of children of a node w. Maintained as part of the representation of T'.

The routines of the algorithm are listed in Figure 5. We initially omit implementation details, in order to establish functional correctness. Some of the implementation details are essential for analyzing the complexity of the algorithm, and will be provided when we pursue that analysis.

```
MakeFSM
```

```
1.  Verify(λ)
2.  PropagateTransitions({ (a, λ) | a ∈ A }, λ)
```

```
Verify(w)
```

```
1.  Set c = head(w), x = tail(w)
2.  Compute ⟨r, u, v⟩ = C_{T'}(x)
3.  If u ≠ λ or v ≠ λ
4.      Insert(r,u,v)
5.      If u ≠ λ
6.          If Traversed[r,head(u)]
7.              Verify(ru)
8.      Else If v ≠ λ and Traversed[r,head(v)]
9.          Verify(rv)
10. Set Tail[w] = pointer to node x
11. Set Transitions[x](c) = w
12. For a ∈ A
13.     If not Traversed[w,a]
14.         Set Traversed[w,a] = true
15.         If w has an edge az in the direction of a
16.             Verify(waz)
```

```
Insert(r, u, v)
```

```
1.  If u == λ
2.      Add r --v--> rv to T'
3.      Set Origin(rv) = Origin(r)
4.  Else
5.      Split r --uy--> ruy into r --u--> ru --y--> ruy
6.      Set Origin(ru) = Origin(r)
7.      Set Traversed[ru,head(y)] = Traversed[r,head(u)]
8.      If v ≠ λ
9.          Add ru --v--> ruv to T'
10.         Set Origin(ruv) = Origin(ru)
```

```
PropagateTransitions(F,w)
```

```
1.  For a ∈ A
2.      If Transitions[w](a) =⊥
3.          Set Transitions[w](a) = F(a)
4.  For v in Children[w]
5.      PropagateTransitions(Transitions[w], v)
```

Figure 5: Algorithm for computing T_{suf}

Proposition 1 `MakeFSM` *constructs* T_{suf}*, and the permanent structure of the associated FSM.*

Proof. We say that `Verify` *visits* a node t of T' whenever the subroutine is invoked with t as its argument. First, we observe that `Verify` visits each node at most once. Clearly, the invocation from Step 1 of `MakeFSM` (see Figure 5) is not repeated. When `Verify` is recursively invoked from its Step 16, the edge leading to the visited node is marked as "traversed," and the node is never visited again from that step. Invocations from steps 7 and 9 visit nodes that have just been created in a call to `Insert`, and whose incoming edges are already marked as traversed. Therefore, `Verify` never revisits a node. Notice also that when new nodes are inserted in the tree (Step 4 of `Verify`), the string associated with the new node is shorter than one that already existed in the tree. It follows from the finiteness of the initial tree T that the total number of nodes inserted is finite, and, thus, the recursion sequence of `Verify` is finite and `MakeFSM` terminates. On the other hand, notice that new nodes that are created are either visited immediately (steps 7 and 9), or their incoming edges were marked as "not traversed." Hence, since the loop in Step 12 recursively traverses all edges outgoing from the current node that had not been traversed (which is done in a conventional pre-order tree traversal recursion), every node of the final tree T' is visited exactly once. We now claim that when the algorithm terminates, $T' = T_{\text{suf}}$. To prove the claim, observe that in Step 1, `Verify` extracts the suffix $x = \text{tail}(w)$ of its argument. In Step 2, the canonical decomposition of x is computed, The first component, r, of this decomposition, corresponds to a prefix of x that is already in the tree. Step 4 constructs the parts of x that were missing. Therefore, a call to `Verify(`w`)` guarantees that $\text{tail}(w)$ will be a node of the constructed tree. Since the algorithm starts with T, and it only adds suffixes of nodes that were already in the tree, every node of T' is either a suffix of a node of T, or a node inserted to allow a bifurcation (e.g., if 001 and 01 are suffixes, a node must exist at 0, even if it is not a suffix). Finally, since all the nodes of T' are visited, every suffix of a node of T is a node of T'. Hence, upon termination of `MakeFSM`, $T' = T_{\text{suf}}$.

Transitions of the FSM associated with T_{suf}, of the form $f(x, c) = cx$, are constructed in Step 11 of `Verify`. Transitions of the form $f(x, c) = u \prec cx$ are added in the `PropagateTransitions` subroutine. Overall, this process exhausts all transitions between permanent states.[13] □

A key supporting structure generated by the algorithm is the array `Tail` of suffix links, constructed in Step 10. A comment is in place here about the apparent redundancy between Step 1 and Step 10, both of which seem to "compute" the longest proper suffix, x, of w. In Step 1, we read the symbols of x *as a substring of* w, a pointer to which we get as input to `Verify`. In Step 10, after possibly having built it, we have access to a *pointer to the node labeled* x in T'. That pointer is then stored in the array `Tail` for use in later stages of the algorithm. Note that w and x, as nodes, could be located in very different parts of T'.

An example of the workings of the algorithm is presented in Figure 6. Figure 6(A) (excluding the dashed arrow) shows a non-FSM GCT T over the alphabet $A = \{\texttt{a},\texttt{b},\texttt{c}\}$. Figures 6(A), 6(B), and 6(C) present the tree T', and the suffix links created, after each iteration of the loop in Step 12 of `Verify(`λ`)` (namely, one iteration for each child of λ in T). Nodes added to T' in each iteration are shaded in light gray, switching to dark gray in later iterations. Nothing changes in the first iteration, except for the addition of the suffix link from node `a`, which is verified in this iteration, to the root. In the second

[13] In addition, also the transitions involving transient states that are nodes of T_{suf} are constructed.

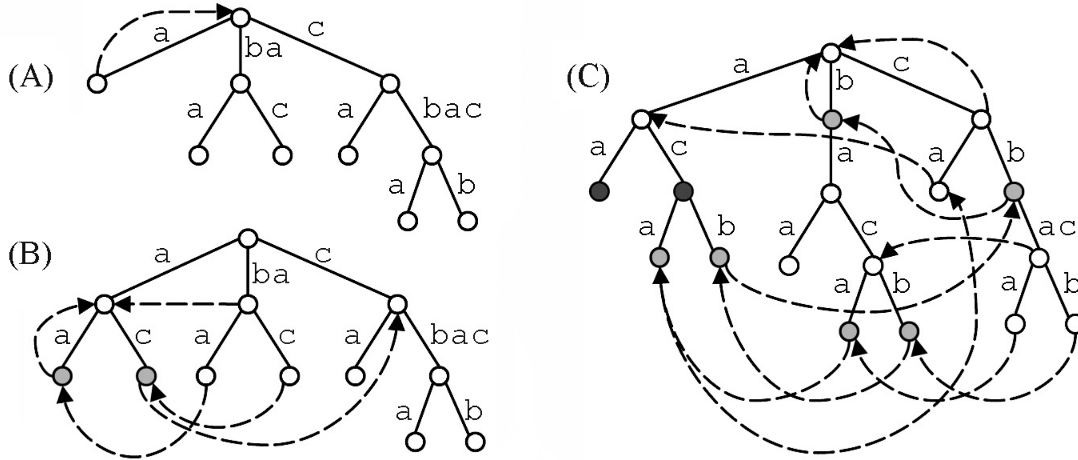

Figure 6: FSM closure (C) and intermediate stages (A,B), with suffix links

iteration, processing the branch in the direction of `b` leads to the verification of nodes `ba`, `baa` and `bac`. The tails of the latter two (`aa` and `ac`, respectively) were not previously in the tree, and are thus added by `Verify` via calls to `Insert`. Since node `a` has already been verified, `Traversed[a,`a`]` is *true* for all $a \in A$ and `Verify` is recursively called for the inserted nodes. No further insertions are required, and suffix links are defined for the new nodes as shown in Figure 6(B). Execution for the branch in the direction of `c` proceeds in a similar way, though in this case, recursive verification of some new nodes leads to further insertions. For instance `Verify(cbacb)` leads to successive creations of nodes `bacb`, `acb`, `cb` and `b`, the latter two causing the split of previously traversed edges. Notice how the search for the longest proper suffix of an inserted node during its verification is helped by following the suffix link of its parent. For example, during verification of `bacb`, we can start the search for `acb` directly at node `ac` by following the suffix link from `bac` in Figure 6(B).

Complexity analysis. Most individual steps of the algorithm listed in Figure 5 can be executed in constant time per node visited, assuming strings associated with edges in the input tree T are efficiently represented, e.g., following the suffix tree methods surveyed in [16], the string v in an edge $u \xrightarrow{v} uv$ of T is defined by a pair of pointers to some memory buffer where the actual symbols are held. Thus, for example, a substring of v can be defined and “copied” somewhere else by manipulating the pointers in constant time. Notice that any new edges inserted by the algorithm are labeled with substrings of previously existing labels, so no additional memory buffer space is needed.

The exception to the statement above is Step 2 of `Verify`, namely, the computation of $C_{T'}(x)$. In principle, the step calls for a string comparison seeming to require symbol by symbol access, and time proportional to $|x|$, which could lead, in the worst case, to total execution time quadratic in the total length of strings in the tree. However, the suffix links available at that point in the execution of the algorithm can be used to improve the efficiency of this computation.

The shallowest layer of executions of the loop in Step 12 iterates over the children of the root λ. It is readily verified, by observing Figure 5, that except for that layer, every time `Verify(`w`)` is invoked,

w is a node of the form $w = auv$, w's parent is $au \neq \lambda$, and au was previously verified and has a suffix link pointing to node u. Then, we can compute $C_{T'}(uv)$ starting from node u, reading individual symbols of v (which we access by reading the appropriate part of w as a string), and traversing the tree until we find the first prefix of uv that is not a word of T'. When the node being verified is $bv \in \text{CHLD}_{T'}(\lambda)$, we take $u = \lambda$, and proceed in a similar fashion with v. In any case, the number of operations is proportional to $|v|$ rather than $|x|$.

Further savings are possible when the invocation `Verify(`ru`)` is made from Step 7. For this case, we show that we can count on $x' = \text{tail}(ru)$ being a word of T', and we only need to find its location in the tree to verify whether it is a node, or an edge must be split to create one. To see this, recall that we get to Step 7 after a call to `Insert` which split an edge $r \xrightarrow{uy} ruy$. Also, since Traversed$[r, \text{head}(u)]$ must be true, the node ruy must have been visited either from Step 16 of `Verify` or from Step 7 or 9 immediately after creation. This guarantees that $\text{tail}(ruy)$ is a node of T' and so $x' = \text{tail}(ru)$ is a word of T'. Now, to find x' in the tree, we can start from node $r' = \text{tail}(r)$, and traverse in the direction of string u, advancing by full edges of the tree, and making comparisons only at the nodes to determine the direction of the next edge. Each time an edge is traversed, we advance in u by the same number of symbols as the length of the edge, until we exhaust the symbols of u. In this case, therefore, the cost of the computation is proportional to the number of nodes we encounter in the path from r' to $r'u$, rather than the number of symbols $|u|$. We will refer to this case as the *fast mode* of `Verify`.

The following theorem bounds the running time of `MakeFSM`. Its proof is based on the observations above, and an analysis of the various configurations arising during the recursive sequence of invocations of routine `Verify`. The proof is deferred to Appendix B.

Theorem 4 *Let $N_E = \sum_e |e|$, where the sum is taken over labels e of edges of T, and let $N' = |T_{\text{suf}}|$, the number of nodes in T_{suf}. Then, `MakeFSM` runs in time $O(N_E + N')$.*

4 Linear time universal coding in the class of tree models

In this section, we apply the results of Section 3 to the implementation, in linear encoding/decoding time, of the semi-predictive approach to universal coding in the sub-class of full-tree models, outlined in Section 1 and reviewed in more detail in the following Subsection 4-A. Thus, we seek universality among full trees only, and the GCT extension is used as an algorithmic tool.

4-A The semi-predictive approach

Next, we review the semi-predictive universal code for full-tree models outlined in Section 1. Notice that the basic idea of a two-pass approach to double universality for countable hierarchies of models is given in [10], the case in which the second pass executes a sequential code is termed semi-predictive in [13], and the class is particularized to full-tree models in [5]. Thus, the universality result presented in this subsection (both for individual sequences and in a probabilistic setting) is well known; it can be found, e.g., in [14], and we re-derive it here for completeness.

For an individual sequence x^n, the coding scheme searches, in a first pass through the data, for the

full tree $T(x)$ that minimizes the (ideal) code length

$$\mathcal{L}(T, x^n) = \mathcal{L}_T^{\mathsf{KT}}(x^n) + \mathcal{C}(T) \tag{14}$$

over all full trees T of any size, where $\mathcal{L}_T^{\mathsf{KT}}(x^n)$ denotes the (ideal) code length assigned by the Krichevsky-Trofimov (KT) sequential probability assignment [15] conditioned on the states of T (with a uniform distribution assigned to symbols occurring in transient states), and $\mathcal{C}(T)$ denotes the cost of encoding T using a *natural* code (see, e.g., [5, 6]). With a natural code, a full tree is encoded with one bit per node, specifying whether the node is a leaf or internal. Thus, by (8),

$$\mathcal{C}(T) = |T| = \frac{\alpha|S_T| - 1}{\alpha - 1}\,. \tag{15}$$

To specify $\mathcal{L}_T^{\mathsf{KT}}(x^n)$, let $n_s(x^{j-1})$ denote the number of occurrences of state $s \in S_T$ in the sequence $s_T(\lambda), s_T(x_1), s_T(x_1x_2), \cdots, s_T(x^{j-1})$, $1 \le j \le n$, and let $n_{s,a}(x^j)$ denote the number of occurrences of $a \in A$ at state s in x^j, namely

$$n_{s,a}(x^j) = |\{i : 1 \le i \le j, x_i = a, s_T(x^{i-1}) = s\}|\,.$$

Clearly, $n_s(x^{j-1}) = \sum_{a\in A} n_{s,a}(x^j)$. Then, upon observing x^j, the KT probability assignment takes the form

$$p_{j+1}(x_{j+1} = a | s_T(x^j) = s) = \frac{2n_{s,a}(x^j) + 1}{2n_s(x^{j-1}) + \alpha}\,.$$

Further, let

$$n(T) = \sum_{s\in S_T} n_s(x^{n-1})\,.$$

Notice that $n - n(T)$ is the number of symbols which, using tree T, are coded in a transient state with a uniform distribution, and is therefore no larger than the depth of T (which, in turn, is at most the number $(|S_T| - 1)/(\alpha - 1)$ of internal nodes of T). Consequently,

$$\mathcal{L}_T^{\mathsf{KT}}(x^n) = (n - n(T)) \log \alpha + \sum_{s\in S_T} \kappa(x^n, s), \tag{16}$$

where hereafter logarithms are taken in base 2, and, by [15],

$$\kappa(x^n, s) \stackrel{\triangle}{=} \log \frac{\Gamma\left(n_s(x^{n-1}) + \frac{\alpha}{2}\right) \Gamma\left(\frac{1}{2}\right)^{\alpha}}{\Gamma\left(\frac{\alpha}{2}\right) \prod_{a\in A} \Gamma\left(n_{s,a}(x^n) + \frac{1}{2}\right)}\,. \tag{17}$$

In a second pass, the algorithm uses the KT probability assignment to encode the data conditioned on $T(x)$. Clearly, upon decoding $T(x)$, the decoder can decode the data in a single pass. Due to the properties of the KT probability assignment [15] used at each state, with an arithmetic coder of sufficient precision, this scheme is twice-universal in the sense that, for any sequence x^n, and *any* number M of states, it achieves a per-symbol code length

$$\mathcal{L}(x^n) = \min_T \mathcal{L}(T, x^n) \le \hat{H}_M(x^n) + \frac{M(\alpha - 1)}{2}\frac{\log n}{n} + O\left(\frac{M}{n}\right), \tag{18}$$

where $\hat{H}_M(x^n)$ denotes the minimum, over all trees with M (permanent) states, of the empirical entropy of x^n conditioned on the tree. Universality in a probabilistic setting also follows by taking expectation with respect to the true model in (18), and noticing that the expected empirical entropy is upper-bounded by the entropy rate.

```
Encode(x^n)
```

```
1.  //First pass:
2.  Compute ST(x), compact suffix tree of x̄^{n-1}$
3.  Compute T'(x) s.t. T'_full(x) = T(x) by pruning ST(x) and making edges leading to leaves atomic
4.  Encode T'_full(x)
5.  //Second pass:
6.  Compute T'_F(x) = T'_suf(x)
7.  Set s = λ
8.  For i in 1..n
9.       Encode x_i using statistics located in array Origin[s] of T'_F(x)
10.      Set s = Transitions[s](x_i) using T'_F(x)
```

Figure 7: Coding algorithm

4-B An efficient algorithm: complexity analysis

We first present the linear time algorithm for the encoding stage. The algorithm is detailed below and summarized in Figure 7.

First encoding pass: finding the optimal tree. A procedure for finding the optimal full tree $T(x)$ in linear time is described in [17]. The procedure described below is similar in that it is also based on the application (pioneered in [21]) of suffix tree techniques, and on the tree pruning ideas of [5]. Nevertheless, our procedure is given in terms of the GCT formalism.

First, notice that all the nodes in $T(x)$ correspond to strings that actually occurred as substrings of $\overline{x}^n$, except for those leaves that are added to complete a full tree, for otherwise $\mathcal{C}(T(x))$ could have been made shorter without affecting $\mathcal{L}^{\mathsf{KT}}_{T(x)}(x^n)$. Let $T'(x)$ denote the GCT that is obtained from $T(x)$ by deleting all leaves that did not occur as substrings of $\overline{x}^n$, as well as any node u such that $\deg(u) = 1$ after deleting those leaves, except if $\overline{u} \preceq x^n$. This exception guarantees that all transient states emitting symbols of x^n take the form $u\$$, where $u \in T'(x)$.[14] Moreover, even though internal nodes of $T'(x)$ may now become permanent states, the only permanent states that will emit symbols of x^n are leaves. Thus, for the purpose of coding x^n, $T'(x)$ is equivalent to $T(x)$. Clearly, we have $T'_{\text{full}}(x) = T(x)$.

Now, consider the *compact suffix tree* (see, e.g., [16]) $ST(x)$ of $\overline{x}^{n-1}\$$ where, as introduced in Section 2, \$ denotes a special symbol that is conceptually assumed to precede x_1. The leaves of $ST(x)$ are given by all strings of the form $\overline{x}^j\$$, $0 \leq j < n$, and v is an internal node of $ST(x)$ if and only if there exist two different symbols $a, b \in A \cup \{\$\}$ such that both $a\overline{v}$ and $b\overline{v}$ are sub-strings of $\$x^{n-1}$ (thus, $\deg(v) > 1$). The last symbol of the string labeling the incoming edge of any leaf of $ST(x)$ is \$ (on the other hand, $T(x)$ is an α-ary tree, and \$ is just a symbol appended to its transient states). The use of this symbol in $ST(x)$ guarantees that the mentioned nodes u of $T'(x)$ with $\deg(u) = 1$ belong also to $ST(x)$. Thus, by the definition of $ST(x)$ and $T'(x)$, all internal nodes of $T'(x)$ are also internal nodes

[14]We recall that, in full generality, transient states correspond to all the words of the tree except for the leaves, and not just to those words corresponding to nodes.

of $ST(x)$. Moreover, the leaves of $T'(x)$ are words of $ST(x)$, but notice that since the incoming edges corresponding to these leaves must be atomic (for otherwise the description of $T(x)$ could be shortened without affecting $\mathcal{L}^{\mathsf{KT}}_{T(x)}(x^n)$), it may be the case that a leaf of $T'(x)$ is not a node of $ST(x)$. Thus, one can obtain $T'(x)$ by "pruning" $ST(x)$ and possibly shortening incoming edges of the resulting leaves to make them atomic (Step 3 in Figure 7). By (16) and (17), the information required for the pruning decisions consists, for each potential leaf s, solely of $\{n_{s,a}(x^n)\}_{a\in A}$. Clearly, these counts are obtained recursively as the sum of the corresponding counts over all children of s. The recursion starts from the leaves $u\$$ of $ST(x)$, for which the symbol a that follows $\$\overline{u}$ in x^n can be recorded during the suffix tree construction and associated to the leaf.

The algorithm that derives $T'(x)$ by pruning $ST(x)$ is based on the observation that, by recursively assigning costs to sub-trees, an optimal tree consists of optimal sub-trees, and can be obtained by dynamic programming. This observation was first made in [5] and is used also in [17]. It should be noticed, however, that the formulation in [5] is simplified by the fact that the tree to be pruned has bounded depth and is atomic. In our case, to assign the costs consider a given GCT T' and a sequence x^n such that for all j, $0 \le j < n$, $s_{T'}(x^j) = s_{T'_{\text{full}}}(x^j)$ (as observed, only sequences emitted from leaves of T' are relevant to the discussion). We associate to each sub-tree rooted at an internal node u of T' the cost $K_{T'}(u)$ recursively defined by

$$K_{T'}(u) = \sum_{w\in\text{CHLD}_{T'}(u)} [K_{T'}(w) + \alpha(|w| - |u| - 1)] + (\alpha - \deg(u)) + \delta_u \log\alpha + 1 \tag{19}$$

where $\delta_u = |\{\, i \,:\, V_{T'}(\overline{x}^i) = u \,\}|$, whereas for a leaf s of T' we define

$$K_{T'}(s) = 1 + \kappa(x^n, s)\,.$$

Since $\mathcal{C}(T'_{\text{full}}) = |T'_{\text{full}}|$, by (14), (15), and (16), we have $K_{T'}(\lambda) = \mathcal{L}(T'_{\text{full}}, x^n)$, as the terms $\alpha(|w|-|u|-1)$ in the summation, and $(\alpha - \deg(u))$ outside the summation on the right-hand side of (19), account for the additional nodes needed to complete T' to a full tree; the term $\delta_u \log\alpha$, on the other hand, accounts for symbols coded in transient states. Thus, Equation (19) can be used as the basis of a dynamic programming minimization procedure. However, the pruning algorithm must also take into consideration the possible insertion of additional nodes in $ST(x)$, as mentioned above, as the incoming edges of the leaves of $T'(x)$ must be atomic.

Specifically, in a post-order traversal [28] of $ST(x)$, we compare, for each node u, the sum of the costs of the optimal sub-trees rooted at all its children, with the cost of making u' a leaf, where $u' \preceq u$ and $|u'| = |\text{PAR}_{ST(x)}(u)|+1$ (or $u' = \lambda$ if $u = \lambda$). It is easy to see that this comparison can be performed by recursively associating to each internal node visited in $ST(x)$ the cost

$$K(u) = \min\left(\alpha(|u| - |u'| + 1) + \sum_{w\in\text{CHLD}_{ST(x)}(u)} K(w)\,,\;\; \kappa(x^n, u)\right) \tag{20}$$

and marking u' as a leaf in case the minimum is achieved by the second argument, where for a leaf $w = v\$$ of $ST(x)$ we define $K(w) = \log\alpha$ in (20). Notice that the term $\delta_u \log\alpha$ of (19) is incorporated into the summation over all children in (20) since, due to the use of the special symbol \$ in $ST(x)$, we have $u\$ \in \text{CHLD}_{ST(x)}(u)$ in case $x^{|u|} = \overline{u}$ (therefore, u may have up to $\alpha + 1$ children).

Summarizing, the computational cost of finding the optimal tree $T(x)$ is given by the cost of the following operations:

1. Building the (compact) suffix tree $ST(x)$ of $\$x^{n-1}$, and some associated data structures;
2. Computing the costs $K(u)$ for all nodes u of $ST(x)$; and
3. Pruning $ST(x)$ in a post-order traversal, with possible insertion of new nodes as leaves of $T(x)$.

It is well known (see, e.g., [16]) that the computational cost of building $ST(x)$ is $O(n)$. The adaptation of the generic suffix tree algorithms to building also some additional *ad hoc* structures (e.g., associating an emitted symbol with each leaf of the tree) is straightforward and does not affect the complexity. Since, by definition, $ST(x)$ has n leaves, it has $O(n)$ nodes when represented as a compact tree. The insertion of additional nodes as possible leaves of $T(x)$ clearly does not affect the linearity. It is shown in [17, Theorem 1] that the computation of each $\kappa(x^n, u)$ can be performed in registers of size $O(\log n)$ in a constant number of operations, and that this precision is sufficient for preserving the validity of (18). Finally, since a post-order traversal of the tree requires a number of operations which is linear in the number of its nodes, the pruning step can also be done in linear time.

Second encoding pass. After encoding $T(x)$ with a natural code (which can be specified recursively with a pre-order traversal of the tree [28]), the encoder makes a second pass through the data which involves, for each j, finding $s_{T(x)}(x^j)$ and arithmetic encoding x^{j+1} using the corresponding KT probability assignment. As observed in [17, Corollary 1], even though $|T(x)|$ may be significantly larger than $|T'(x)|$ (since $ST(x)$ is a compact tree), it is still $O(n)$, for otherwise $T(x)$ would not have emerged as the optimal tree in (14) (think, e.g., of the tree $\{\lambda\}$, for which $\mathcal{C}(\{\lambda\}) = 1$ and $\mathcal{L}^{\mathsf{KT}}_{\{\lambda\}}(x^n) < n \log \alpha + o(n)$). Therefore, $T(x)$ can be described in linear time. As for arithmetic coding once $s_{T(x)}(x^j)$ is determined, it is shown in [9] that, again, performing a constant number of arithmetic operations per symbol in registers of size $O(\log n)$ guarantees a precision that will not affect the validity of (18). Thus, we focus on the determination of the state.

In [17], the BWT of $\$x^n$ facilitates the transition between states in constant time. Alternatively, notice that the algorithms that construct $ST(x)$ in linear time can also maintain pointers (the so-called *suffix links*) between each leaf $au\$$ of $ST(x)$, and the leaf $u\$$, as suffixes are inserted by length. If each leaf $u\$$ is in turn linked to the corresponding state $s_{T(x)}(\overline{u})$, then each state transition can be done in constant time. Clearly, these links can be created with an additional traversal of $ST(x)$ in linear time, or during the pruning phase without affecting its complexity. These methods, however, require either the BWT of $\$x^n$, or the suffix tree $ST(x)$, none of which are, in principle, available to the decoder. Thus, we propose an alternative linear time method, based on the FSM closure of $T'(x)$, that can be employed also at the decoding side.

Specifically, before starting the second pass, the encoder builds an FSM closure $T'_F(x)$ of $T'(x)$ (without loss of generality, $T'_{\text{suf}}(x)$), using the algorithm `MakeFSM` of Section 3-D.[15] For every permanent state w of $T'_F(x)$, and every symbol $c \in A$, the encoder then has access to the next-state transition $f(w, c)$ via the mapping `Transitions[`w`]`. This mapping also provides the state transitions for all

[15] Notice that while only states associated with leaves actually occur in the sequence of permanent states of $T'(x)$ determined by x^n, this is no longer the case with $T'_F(x)$, for which we take full advantage of the GCT formalism.

transient states that are associated with nodes. Since, by the definition of $T'(x)$, all the transient states that are actually visited with x^n are indeed associated with nodes, it follows that, starting from the root (which is used to encode x_1), the encoder can make each transition between states of $T'_F(x)$ in constant time. In addition, the link `Origin[`w`]` provides access to the state of $T'(x)$ that is being refined by w, which accumulates the relevant statistics for the KT probability assignment (loop starting at Step 8 in Figure 7). These statistics are possibly shared with other states of $T'_F(x)$. The following corollary to Theorem 4 establishes the linear time complexity of the proposed encoder.

Corollary 1 *`MakeFSM`$(T'(x))$ runs in time $O(n)$.*

Proof. By Theorem 4, it suffices to prove that both the sum of the lengths of the strings that label edges of $T'(x)$ and the number of nodes of $T'_F(x)$ are $O(n)$. The former is clearly upper-bounded by $|T(x)|$, which was already observed to be $O(n)$. As for $T'_F(x)$, let $T''(x)$ denote the tree obtained by deleting from $T'(x)$ those nodes that are not in $ST(x)$ (and the corresponding incoming edges). By the definition of a (compact) suffix tree, $u \in ST(x)$ if and only if either $\overline{u}$ is a prefix of $\$x^{n-1}$, or there exist $a, b \in A \cup \{\$\}$, $a \neq b$, such that both $a\overline{u}$ and $b\overline{u}$ are sub-strings of $\$x^{n-1}$. Thus, every suffix of a node of $ST(x)$ is also a node of $ST(x)$. Since $T''(x) \subseteq ST(x)$, and $T''_F(x)$ is formed by adding as nodes all the suffixes of the nodes of $T''(x)$, it follows that the added nodes are in $ST(x)$, and therefore we also have $T''_F(x) \subseteq ST(x)$. Consequently, $|T''_F(x)| = O(n)$. Now, in addition to the nodes of $T''_F(x)$, $T'_F(x)$ includes all the suffixes of the nodes of $T'(x)$ that are not in $ST(x)$. As observed in the description of the pruning step, these nodes can only be leaves of $T'(x)$, and all corresponding incoming edges have length 1. Thus, these leaves take the form wa, where $w \in T''(x)$ (and, hence, $w \in T''_F(x)$) and $a \in A$. Thus, the corresponding suffixes take the form va, where $v \in T''_F(x)$, so that the number of additional nodes of $T'_F(x)$ cannot be larger than $\alpha|T''_F(x)| = O(n)$. □

Decoding. The situation would be analogous at the decoder if, as it starts scanning the compressed bit-stream, it had access to $T'(x)$. However, only $T(x)$ has been described and, by Theorem 3, its FSM closure might, in principle, have a super-linear number of nodes. Of course, a modified encoder can describe $T'(x)$ (by simply specifying, for every node of $T(x)$, whether it is also a node of $T'(x)$), without affecting the validity of (18). This modified code is still universal, and a linear time implementation of the decoder follows trivially from reversing some of the operations at the encoder. However, it is not necessary to penalize the code length to preserve linear time complexity. Next, we present a decoder that has access to $T(x)$ only, but requires a more elaborate analysis.

Assume $T(x) \neq \{\lambda\}$ (for otherwise $T'(x)$ would also be known), and let $\hat{T}'(x)$ denote the tree obtained from $T(x)$ by deleting all the leaves, as well as nodes whose outgoing degree after deleting the leaves is 1. Clearly, $\hat{T}'_{\text{full}}(x) \subseteq T(x)$ (equality holds only for $T(x) = \lambda$), and $|\hat{T}'(x)| \leq |T'|$ for any T' such that $T'_{\text{full}} = T(x)$. Thus, $\hat{T}'(x) \subseteq T'(x)$. The decoder starts by building an FSM closure $\hat{T}'_F(x)$ of $\hat{T}'(x)$, which, by the proof of Corollary 1, can be done in linear time (again, we assume $\hat{T}'_F(x) = \hat{T}'_{\text{suf}}(x)$). Then, the key idea is to relate, for every i, $0 \leq i < n$, the state $\hat{s}_i \triangleq s_{\hat{T}'_F(x)}(x^i)$ to $s_i \triangleq s_{T_F(x)}(x^i)$, which is the state needed by the decoder, and to show that the linkage between the two states can be executed in linear time. The statement of this relation and the complexity analysis of the procedure are deferred to Appendix C.

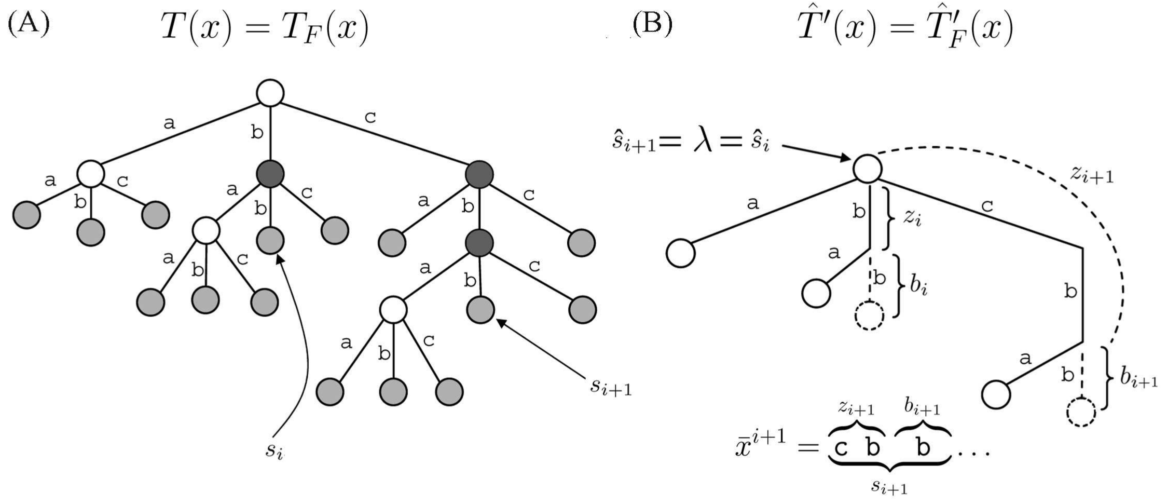

Figure 8: Decoding tree

Figure 8 illustrates some of the decoding operations. Figure 8(A) shows a tree $T(x)$, and the leaves (light gray) and internal nodes (dark gray) deleted to obtain $\hat{T}'(x)$. The latter tree is shown in Figure 8(B), which also illustrates the relation between $\hat{s}_i$ and s_i. In this particular case both $T(x)$ and $\hat{T}'(x)$ are FSM. The strings z_i and symbols b_i shown in the figure are discussed in Appendix C.

The implementation of the semi-predictive approach to Context algorithm outlined in this section (and in Appendix C) is denoted `SPContextFSM`. The following theorem summarizes our discussion.

Theorem 5 *`SPContextFSM` encodes and decodes any sequence x^n in time $O(n)$.*

Remarks

(a) `SPContextFSM` does not explicitly obtain $T'(x)$, the pruned tree given by the set of nodes of $T(x)$ that actually occurred as substrings of $\overline{x}^n$. Since, after decoding x^n, the decoder can determine $T'(x)$, a plausible approach to linear time decoding would be to obtain $T'_F(x)$ adaptively, starting from $\hat{T}'_F(x)$ and adding the missing nodes as new words of $T(x)$ are decoded. Thus, $\hat{T}'_F(x)$ would grow on the fly "as needed." It can be shown that such a procedure can indeed be implemented in linear time without recourse to additional data structures such as `Jump[u]`. However, the description of `SPContextFSM` is simpler.

(b) `SPContextFSM` is presented as an application of the concept of FSM closure, solving the open problem of linear time decoding. In [39], we present another solution to this problem, without recourse to the FSM closure. This approach will typically require more storage space than `SPContextFSM` at the decoder. The idea in [39] is to extend $T(x)$ on the fly with the suffix tree of the string decoded so far. To this end, the suffix tree of $\overline{x}^{i-1}\$$ is built in an "anti-sequential" manner as in [20], i.e., the suffix tree of $\overline{x}^j\$$ is available at each step j, $0 < j < i$.

4-C Context Algorithm and the BWT

Algorithm `SPContextFSM` sheds light on the relation between the *Context* algorithm and coding schemes based on the BWT [19], which use similar data structures. An information-theoretic analysis of this transform in the context of tree models was first given in [22]. The discussion below will show that the sub-optimality of BWT-based codes in compressing tree models is the result of inverting the order in which operations are carried out in `SPContextFSM`, without any significant complexity advantage.

The BWT traverses the suffix tree $ST(x)$ sorting the suffixes of x^n lexicographically, and outputting the symbol emitted at each suffix. Thus, it reorders x^n so that symbols that occurred in "similar" contexts are grouped together for *any* tree model structure T, without attempting at selecting the best coding tree. Coding is then typically performed by move-to-front heuristics, or by segmenting the (transformed) sequence sub-optimally [22], and treating each segment as an i.i.d. sequence. Thus, the statistics of nearby contexts are merged. `SPContextFSM`, instead, groups statistics only among symbols that occurred at the same context in the optimal tree $T(x)$. As shown in Section 4-B, the complexity of finding $T(x)$ (by pruning $ST(x)$) and of building the FSM closure of the equivalent GCT $T(x)$ is similar to that of implementing the BWT and, in fact, the schemes use similar algorithmic tools. Moreover, the decoder need only build an FSM closure, with a complexity that depends on the size of the optimizing tree. Even though this complexity might be $O(n)$ in the worst case, it will typically be much smaller in practice. Then, the decoder can proceed to decode the sequence on the fly. In contrast, in BWT-based schemes, the decoder typically decodes the entire sequence, and only then proceeds to reorder it by performing the inverse BWT.

Yet, the "context deinterleaving" feature of BWT (in which symbols emitted in the same context are processed consecutively) may still be desirable in situations in which we want to avoid frequent context switching. However, this property is provided in a more efficient manner by an *invertible* transform of the input sequence, defined for each tree T, which we will call the T-transform. Specifically, for a GCT T, we order the permanent states, say, lexicographically, and we reorder a sequence x^n so that all symbols emitted at the same state are contiguous and appear in chronological order in a "segment," with segments ordered by the lexicographical order of the states, preceded by all the symbols emitted in transient states (also in chronological order). By following the state transitions *of the FSM built from* T, each input symbol is assigned to the appropriate segment. Clearly, given all segment lengths, the T-transform is invertible since, given x^i, x_{i+1} is the first symbol in the segment corresponding to $s_{T(x^i)}$ that has not yet been inverted. Moreover, just as with the BWT, the transform and its inverse can be implemented in linear time with the tools described in this paper. Selecting $T = T(x)$, each segment can be coded separately with a memoryless model. If the coding scheme starts by describing $\alpha-1$ symbol counts for the segment (rendering the overall approach two-pass, rather than semi-predictive), the length information necessary for inverting the T-transform can be omitted, as it is implicit in the type information. With an enumerative code for each segment type [40], such schemes are known to yield the same code length as Laplace's rule of succession, and are therefore still optimal in a probabilistic setting (but, as opposed to the KT rule, not for individual sequences; see, e.g., [11]).[16] An

[16] We point out that the T-transform extends and formalizes ideas that are related to [23]. In [23], length information is also transmitted, but the pruning procedure (that yields a possibly incomplete tree) is heuristic. It can be interpreted as an attempt at optimizing the code length in the GCT class.

Sequence x^n:	1 1 0 0 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 0 1 0
States s_0^{n-1}:	λ β D A A A A B A C D A A B A C D A A B A C D A A B A A B
Transformed sequence:	1 1 0 0 0 1 1 0 1 1 0 1 1 0 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0
States:	λ β A A A A A A A A A A A A A A A B B B B B C C C D D D D

Figure 9: T-transform of a binary sequence

example of the T-transform of a binary sequence x^n, whose optimal context tree $T(x)$ is T of Figure 1, is given in Figure 9. In the figure, the states $\lambda\$, 1\$, 0, 100, 101$, and 11 of T (with permanent states in lexicographical order) have been re-labeled λ,β,A,B,C, and D, respectively, for succinctness. The T-transform is computed by means of the FSM closure of T, which is given in automaton form in Figure 4.

One situation in which it may be advantageous to group together symbols occurring in the same context arises when we wish to replace the arithmetic code with a simpler "symbol-by-symbol" code on an extended alphabet.[17] Even though arithmetic coding does not affect the linearity of the complexity, it is sometimes considered, in practice, an expensive operation. Obviously, the same simple symbol-by-symbol coding techniques that are used in BWT-based schemes can be employed with the T-transform. Of course, in this case, the cost function to be minimized in the pruning of $ST(x)$ should account for the specific code adopted, rather than for the KT code length.

To summarize our discussion, we notice that the description of the T-transform reveals, in fact, the main weakness of the BWT approach to universal lossless compression. While the approach with the T-transform is to first prune the suffix tree to obtain the best coding tree, and then reorder the input sequence, BWT first reorders the input sequence in a manner that is compatible with *any* coding tree, and then the transformed sequence is segmented by *ad hoc* (or sub-optimal, see [22]) means.

5 Tree models under time reversal

In this section, we investigate the effect of time reversal on the minimal tree model of a finite-memory process. The investigation is motivated by the question posed in Section 1 of possible differences in compression performance between a left-to-right and right-to-left scanning of a string by a universal compressor. Since the effect we are most interested in is the possible variation in estimated model size, we restrict our attention to full-tree models, for which efficient model optimization algorithms exist. As mentioned, this is an open question for the more general GCT models. *All trees mentioned in the remainder of the section are assumed to be full α-ary trees.* We shall also assume that in all finite-memory processes mentioned, the defining CPMFs associated with permanent states of a GCT are such that any Markov chains associated with these states are *irreducible* and *aperiodic* [27, 24].

[17] In fact, such deinterleaving techniques date back to [41] and were crucial, e.g., for facsimile coding, before the invention of arithmetic coding. In [41], an interlacing scheme for taking separate extensions for each state in the binary case is proposed.

It is known (see, e.g., [24, Ch. 4]) that the order and entropy rate of a stationary Markov process are preserved under time reversal. However, the translation of this fact to the formal setting of string processes defined in Section 2 involves some nuances, including an appropriate characterization of stationarity in that setting. In any case, the classical results do not address the issue of the effects of time reversal on the more detailed tree structure or its size. As in Section 4, the FSM closure will play a crucial role in the derivation of the main results of this section.

5-A Two-sided finite-memory processes

We define the *reverse* of a string process P as a probability assignment $\overline{P} : A^* \to [0,1]$ such that

$$\overline{P}(u) = P(\overline{u}), \quad u \in A^*.$$

Clearly, since P satisfies postulate **(Q1)** of Section 2, so does $\overline{P}$. For $\overline{P}$ to satisfy **(Q2)**, P must satisfy

($\overline{\mathbf{Q2}}$) $P(u) = \sum_{a \in A} P(au), \quad \forall u \in A^*.$

Therefore, a process satisfying **(Q1)**, **(Q2)**, and ($\overline{\mathbf{Q2}}$) is reversible, in the sense that its reverse is also a process. We call such a process *two-sided.*

For a tree T with a set of leaves S_T and a string u over A, define

$$S_T[u] = \{\, v \in S_T \mid u \preceq v \,\}.$$

Lemma 11 *Let r be an arbitrary node of a tree T. If P is any two-sided process, then, for all $u \in A^*$, we have*

$$\sum_{v \in S_T[r]} P(\overline{v}u) = P(\overline{r}u). \tag{21}$$

Proof. Let T_r denote the subtree of T rooted at r. We prove the claim by induction on $|T_r|$. It holds trivially for for all r such that $|T_r| = 1$. Assume it holds for all r' such that $|T_{r'}| < |T_r|$, and let $T_{ra},\ a \in A$, denote the subtree rooted at ra. For $v \in S_T[ra]$, write $v = rav'$. Then, we have

$$\sum_{v \in S_T[r]} P(\overline{v}u) = \sum_{a \in A} \sum_{v \in S_T[ra]} P(\overline{v}u) = \sum_{a \in A} \sum_{v' \in S_{T_{ra}}} P(\overline{v}' a \overline{r} u) = \sum_{a \in A} P(a\overline{r}u) = P(\overline{r}u),$$

where the next to last equality follows from the induction hypothesis applied to the root of the subtrees T_{ra}, and the last equality follows from ($\overline{\mathbf{Q2}}$). □

Recall from Section 3-B that if $\langle T, p\rangle$ is a full-tree model of a process, and the GCT T_F is an FSM refinement of T, then T_F is an extension of T (Lemma 6), and $\langle T_F, p'\rangle$, with $p'(\cdot|t) = p(\cdot|s),\ s \in S_T,\ t \in S_{T_F}[s]$, generates, by Lemma 3, the same process as $\langle T, p\rangle$. We say that the CPMF set $\{p'\}$ is derived *by refinement* from $\langle T, p\rangle$.

Lemma 12 *Assume $\langle T, p\rangle$ generates a two-sided finite-memory process P of order m. Then,*

(i) *$\overline{P}$ is a finite-memory process of order m, and*

(ii) *if T_F is an FSM refinement of T, and $p_F(t)$ denotes the stationary probability of $t \in S_{T_F}$ in the Markov chain defined by the permanent states of T_F, with transition probabilities derived by refinement from $\langle T, p\rangle$, then P satisfies*

$$P(\overline{s}) = \sum_{t \in S_{T_F}[s]} p_{_F}(t) \quad \forall s \in S_T. \tag{22}$$

Proof. (i) If such a process exists, then the finite-memory property of $\overline{P}$ is an immediate consequence of that of P. Let $\overline{z} \in A_P^m$, and $a \in A$. Then, for all u such that $\overline{z}\,\overline{u} \in A_P^*$, we have

$$\overline{P}(a|uz) = \frac{\overline{P}(uza)}{\overline{P}(uz)} = \frac{P(a\overline{z}\,\overline{u})}{P(\overline{z}\,\overline{u})} = \frac{P(a\overline{z})P(\overline{u}|a\overline{z})}{P(\overline{z})P(\overline{u}|\overline{z})} = \frac{P(a\overline{z})P(\overline{u}|\overline{z})}{P(\overline{z})P(\overline{u}|\overline{z})} = \frac{P(a\overline{z})}{P(\overline{z})}, \tag{23}$$

where we have extended the conditional probability notation to strings in a natural manner. Since the right-most side of (23) is independent of u, the order $\overline{m}$ of $\overline{P}$ is at most m. Considering now the process $\overline{P}$, it follows that the order of its reverse is at most $\overline{m}$. Since $P = \overline{\overline{P}}$, we conclude that $m = \overline{m}$.

(ii) If $|T| = 1$, there is only one CPMF in play and the claim is straightforward. Hence, we assume $|T| > 1$. Since T_F is an extension of T, $S_{T_F}[s]$ is not empty for any $s \in S_T$, and by Lemma 11 applied to the subtree of T_F rooted at s, with $u = \lambda$, we have

$$P(\overline{s}) = \sum_{t \in S_{T_F}[s]} P(\overline{t}). \tag{24}$$

For a state $t \in S_{T_F}$, write $t = br$, $b \in A$. Since T_F is FSM, by Theorem 2, we have $r \in T_F$, and $S_{T_F}[r]$ is not empty. In fact, $S_{T_F}[r]$ is the set of permanent states of T_F that have FSM transitions to t. Applying Lemma 11 to the subtree rooted at r, and with $u = b$, we can write

$$P(\overline{t}) = \sum_{z \in S_{T_F}[r]} P(\overline{z}b) = \sum_{z \in S_{T_F}[r]} P(\overline{z})p'(b|z), \qquad t \in S_{T_F}. \tag{25}$$

Here, $p'(\cdot|z)$ is the CPMF derived by refinement from $\langle T, p\rangle$. Recalling that z runs over all states with transitions to t, we recognize (25) as a typical equation in the linear system satisfied by the stationary probabilities of the Markov chain induced by P, with set of states S_{T_F}. Under our assumptions on the processes of interest, this system has a unique solution, $\{\, P(\overline{t}) = p_{_F}(t) \,|\, t \in S_{T_F} \,\}$. Part (ii) of the lemma now follows from (24). □

Lemma 12 determines the probability assignments for strings corresponding to permanent states of T as a function of the stationary probabilities of the permanent states of an FSM refinement of T. It can be readily verified that these assignments are, in fact, independent of the specific FSM refinement selected. This independence is due to the fact that all FSM refinements are refinements of the FSM closure (see Lemma 7), which is, in turn, a refinement of T, and the CPMFs are propagated by refinement. The lemma also shows that if $\langle T, p\rangle$ generates a two-sided finite-memory process, then the probability assignments for (short) strings that do not select permanent states are uniquely determined by the permanent CPMFs, and must be obtained as marginals of the (stationary) probabilities of the

permanent states (via Lemma 11).[18] This observation also tells us how to construct a two-sided finite-memory process given its permanent CPMFs. The discussion is summarized in the following theorem.

Theorem 6 *Let T be a tree of depth m, and $\{p(\cdot|s)\,|\,s \in S_T\}$ a set of CPMFs associated with the permanent states of T, such that the Markov chain associated with $T_{\rm suf}$ by refinement is irreducible and aperiodic. Then, there exists one and only one assignment of CPMFs to transient states of T such that the defined finite-memory process P is two-sided. The resulting reverse $\overline{P}$ is a finite-memory process of order m.*

Remarks. The results above show that any process that satisfies **(Q1)**,**(Q2)**, ($\overline{\mathbf{Q2}}$), and the finite-memory property is a two-sided tree process whose reverse is also a tree process. In fact, it turns out that these requirements are redundant: it can be readily shown that any process that satisfies **(Q1)**,($\overline{\mathbf{Q2}}$), and the finite-memory property must also satisfy **(Q2)**.

The relation between condition ($\overline{\mathbf{Q2}}$) and the stationarity of the process P was observed in [25]. It is known that time reversal preserves the entropy of a stationary Markov chain (see, e.g., [24]). In the string process setting, a comparison of entropies is meaningful only after restricting the string probability assignment to a domain where it defines a random variable. As noted in [25], a string process P defines a random variable if and only if it is restricted to a full prefix-free subset of A^*. For this condition to hold for both P and $\overline{P}$, the subset has to be both prefix- and suffix-free, e.g., the set of all strings of a given length n (other such *fix-free* sets are possible [42]). Once this formality is satisfied, it is obvious that P and $\overline{P}$ define random variables of the same entropy. In addition, P and $\overline{P}$ have the same order, as stated in Lemma 12. This fact constrains their minimal tree representations to be of the same maximum depth (see Lemma 2), but does not say much more about relations between their sizes or structures. We investigate these questions next.

5-B Reverse trees

It follows from Theorem 6 that $\overline{P}$ admits a minimal tree model $\langle \widehat{T}_p, \hat{p} \rangle$. One explicit method to compute the CPMFs $\hat{p}(\cdot|t),\ t \in S_{\widehat{T}_p}$ in terms of the parameters of $\langle T, p \rangle$ proceeds by first extending T to a full balanced tree T_F of depth m (which is an FSM refinement of T), extending the CPMFs $p(\cdot|\cdot)$ by refinement accordingly, then solving the system (25) for the stationary probabilities $P(\bar{t}),\ t = t_1^m \in S_{T_F}$, and finally writing

$$\hat{p}(a|\bar{t}) = \frac{\overline{P}(ta)}{\overline{P}(t)} = \frac{P(a\bar{t})}{P(\bar{t})} = \frac{P(a\overline{t_2^m})p(t_1|a\overline{t_2^m})}{P(\bar{t})}, \quad \bar{t} \in S_{T_F}. \tag{26}$$

To obtain a minimal tree $\widehat{T}_p$, by Lemma 2, sets of identical sibling CPMFs are merged recursively to the maximum possible extent. The tree $\widehat{T}_p$ in this model depends on both T and p. In contrast, we are interested in the minimal representation for the reverses of *all* the processes whose minimal tree models have T as underlying graph, and we define the *reverse tree* of T as

$$\overline{T} = \bigcup_{p(\cdot|\cdot) \in \mathcal{P}} \widehat{T}_p\ ,$$

[18]It is easy to see that this assignment is consistent with (3).

which depends solely on T. The union is taken over the set $\mathcal{P}$ of all CPMFs $p(\cdot|\cdot)$ for which P is irreducible and aperiodic. The tree $\overline{T}$ is the minimal common refinement of all $\widehat{T}_p$ for the given tree T. Notice that, while there is a symmetry between T and $\widehat{T}_p$, so that $\widehat{(\widehat{T}_p)}_{\hat{p}} = T$, no such symmetry exists between T and $\overline{T}$, and we might have $\overline{\overline{T}} \neq T$. We will use $\overline{T}$ as a tool to bound the size difference between T and $\widehat{T}_p$, which is what matters when dealing with a specific process P.

We can also view $\overline{T}$ as the minimal tree of a reversed process $\overline{P}_\xi$, where P_ξ is a *symbolic process* with a minimal tree model $\langle T, p_\xi \rangle$ in which we have substituted $(\alpha-1)$ symbolic indeterminates $\xi_{a,s}$ for the free parameters $p(a|s)$ at each state s. The conditional probabilities of the symbolic reversed process, obtained, as before, from (26) and the system (25), are rational functions in the indeterminates $\xi_{a,s}$. Following the initial computation of the parameters of the full balanced tree, the structure of $\overline{T}$ emerges through recursive merging of sets of sibling leaves with identical symbolic CPMFs. At the end of this process, every set of sibling leaves contains at least two CPMFs that are not identical vectors of rational functions. A *specific* (numerical) assignment p of CPMFs can still lead to further merging of leaves, resulting in $\widehat{T}_p \subset \overline{T}$. However, this can happen only when p (regarded as a vector in $\mathbb{R}^{(\alpha-1)|S_T|}$) lies in a nontrivial algebraic variety, namely, the set of solutions of the nontrivial system of polynomial equations in the $\xi_{a,s}$ obtained by imposing equality on sets of sibling CPMFs. This algebraic variety has measure zero in $\mathcal{P}$ (which has positive measure in $\mathbb{R}^{(\alpha-1)|S_T|}$, since CPMF assignments that do not satisfy the irreducibility and aperiodicity conditions also form a zero-measure set). By a similar argument, given a tree T, the set of CPMFs p such that $P(u)$ vanishes for some u also has measure zero in $\mathcal{P}$. The foregoing discussion is summarized in the following lemma.

Lemma 13 *For any tree T, and almost all CPMFs $p(\cdot|\cdot) \in \mathcal{P}$, we have $\widehat{T}_p = \overline{T}$, and $P(u) \neq 0$ for all $u \in A^\star$.*

The tree T of the example in Figure 1 is shown again in Figure 10(A), with a symbolic parametrization. The reverse tree $\overline{T}$ and its associated conditional probabilities obtained by explicitly solving the system of symbolic equations resulting from (25) and (26) are shown in Figure 10(B). Since, in the example, $\alpha = 2$, we omit the first index in $\xi_{a,s}$ after assuming $a = 0$. The conditional distributions associated with the nodes of $\overline{T}$ are, as expected, rational functions of the ξ_s. Thus, $\overline{T}$ is associated with exactly the same number of free parameters as T, even though it has more leaves. These "hidden" redundancies would not be exploited efficiently by modeling algorithms that target the class of tree models (e.g., Context, CTW).

The symbolic procedure outlined above will produce the reverse tree $\overline{T}$ and its symbolic parametrization for any given tree T. However, the procedure does not provide general insight into the structure of $\overline{T}$. To this end, we now derive a combinatorial characterization and construction of $\overline{T}$.

Given a tree T, let $R(T)$ be the smallest full α-ary tree constructed according to the following rule: *For every internal node w of T, $\overline{w}$ is an internal node of $R(T)$.* Notice that $R(T)$ might contain internal nodes that are added to satisfy structural constraints of the tree, rather than directly as reverses of internal nodes of T. However, any internal node of $R(T)$ that has only leaves as children must be the reverse of an internal node of T (as its children must have been included in $R(T)$ to make the node internal, and not as reverses). We will rely on this fact in the propositions below.

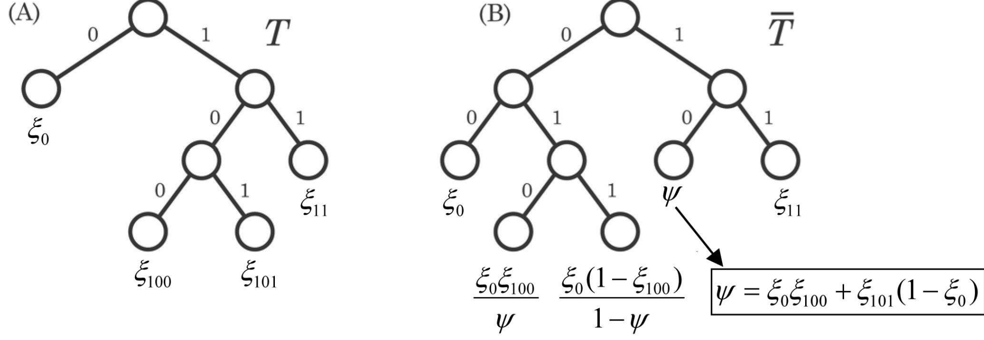

Figure 10: Binary tree with symbolic parametrization and reversed tree

Theorem 7 *For every tree T, we have $\overline{T} = R(T)$.*

Proof. Assume $\overline{T} \not\subseteq R(T)$. Then, there exists an internal node $s \in \overline{T}$ such that $sa \in S_{\overline{T}}$, $a \in A$, and s is not an internal node of $R(T)$. By the definition of $R(T)$, $\overline{s}$ is not an internal node of T, which implies that it selects a permanent state of T. Choosing, by Lemma 13, $p(\cdot|\cdot) \in \mathcal{P}$ such that $\widehat{T}_p = \overline{T}$ and $P(u) \neq 0$ for all $u \in A^*$, it follows that $P(a|bs)$ is independent of b for all $a, b \in A$. Thus,

$$\hat{p}(b|sa) = \frac{\overline{P}(a\overline{s}b)}{\overline{P}(a\overline{s})} = \frac{P(bsa)}{P(sa)} = \frac{P(bs)P(a|bs)}{P(s)P(a|s)} = \frac{P(bs)P(a|s)}{P(s)P(a|s)} = \frac{P(bs)}{P(s)}\,,$$

which does not depend on a. As a result, leaves sa of $\overline{T} = \widehat{T}_p$ share the same CPMF, contradicting the minimality of $\widehat{T}_p$.

Assume now that $R(T) \not\subseteq \overline{T}$. Then, there exists a leaf $s \in S_{R(T)}$ such that $s \notin \overline{T}$. Write $s = tua$, where $t \in S_{\overline{T}}$, $a \in A$, and $u \in A^*$. Since tu is an internal node of $R(T)$, there exists an internal node $tv \in R(T)$ all of whose children are leaves of $R(T)$. By the property discussed before Theorem 7, $\overline{v}\,\overline{t}$ must be an internal node of T. Define a tree T', obtained from T by pruning all nodes descending from, but not including, the children of $\overline{v}\,\overline{t}$, thus making the latter leaves of T'. Let P' be a process for which T' is a minimal tree. Since $T' \subseteq T$, T can generate all the processes that T' generates, and, therefore, we must also have $\overline{T}' \subseteq \overline{T}$. Hence, there exists a prefix $w \preceq t$ such that w is a leaf of $\overline{T}'$. Write $tv = wr$. Then, for all $b \in A$, we have

$$P'(b|awr) = \frac{P'(awrb)}{P'(awr)} = \frac{\overline{P}'(a|b\overline{r}\,\overline{w})P'(wrb)}{P'(awr)} = \frac{\overline{P}'(a|\overline{r}\,\overline{w})P'(wrb)}{P'(awr)} = \frac{P'(wrb)}{P'(wr)},$$

where the third equality follows from the fact that w is a leaf of $\overline{T}'$. We get $P'(b|awr)$ independent of a, contradicting the fact that $\overline{r}\,\overline{w}a = \overline{v}\,\overline{t}a$, $a \in A$ are leaves of the minimal tree T'. □

It follows from Theorem 7 and the definition of $R(T)$ that $\overline{T}$ has the same depth as T, and that when T is a full balanced tree, so is $\overline{T}$, consistent with the time reversal properties of stationary Markov processes [24]. The following lemmas establish a connection between $R(T)$ and the FSM closure of T, which will allow us to bound the size of $R(T)$. The first lemma below shows that if T is FSM, then all nodes of $R(T)$ are there to satisfy its basic construction rule directly.

Lemma 14 *If T_F is FSM, then v is an internal node of $R(T_F)$ if and only if $\overline{v}$ is an internal node of T_F. Therefore, $|R(T_F)| = |T_F|$, and $R(R((T_F))) = T_F$.*

Proof. It suffices to prove that if $sa \in R(T_F)$ is the reverse of an internal node of T_F, then so is s. Let $a\overline{s}$ be an internal node of T_F, so that $sa \in R(T_F)$. Then, $a\overline{s}\,\overline{v}$ is a leaf of T_F for some $v \neq \lambda$, and, by the FSM property of the full tree T_F and Theorem 2, we have $\overline{s}\,\overline{v} \in T_F$. Hence, $\overline{s}$ is an internal node of T_F. □

Lemma 15 *For any tree T, $R(T)$ is FSM, $R(T) = R(T_{\mathrm{suf}})$, and $|R(T)| = |T_{\mathrm{suf}}|$.*

Proof. Consider a leaf $s = au$ of $R(T)$. If $u = \lambda$, then $u \in R(T)$. Otherwise, $s = avb$ for some $b \in A$, and, thus, there exists a node $avw \in R(T)$ all of whose children are leaves. Such a node must be in $R(T)$ only because $\overline{w}\,\overline{v}a$ is an internal node of T, and, hence, $\overline{w}\,\overline{v}$ is also an internal node of T, which, in turn, implies by construction that vw is an internal node of $R(T)$, and so is v. Therefore, vb is a node of $R(T)$. It follows that for all $s \in S_{R(T)}$, $\mathrm{tail}(s)$ is a node of $R(T)$, which, by Lemma 8, implies that $R(T)$ is FSM.

We claim that if T_1 and T_2 are trees such that $T_1 \subseteq T_2$, then $R(T_1) \subseteq R(T_2)$. If u is an internal node of $R(T_1)$ then $\overline{v}\,\overline{u}$ is an internal node of T_1 for some $v \in A^*$, i.e., either u is the reverse of an internal node of T_1, or was added to $R(T_1)$ to allow the insertion of a descendant node uv with that property. Therefore, $\overline{v}\,\overline{u}$ must be an internal node of T_2 (it cannot be a leaf of T_2, since otherwise its children would not be in T_2, contrary to the inclusion assumption). Thus, $uv \in R(T_2)$, and, hence, $u \in R(T_2)$. Clearly, if all the internal nodes of $R(T_1)$ are internal nodes of $R(T_2)$, then all the leaves of $R(T_1)$ must be nodes of $R(T_2)$. Thus, $R(T_1) \subseteq R(T_2)$, as claimed. Therefore, $R(T) \subseteq R(T_{\mathrm{suf}})$. Consider now an internal node $u \in R(T_{\mathrm{suf}})$. By Lemma 14, $\overline{u}$ is an internal node of T_{suf}, so $\overline{u}b \in T_{\mathrm{suf}}$, and $\overline{u}b$ is a suffix of a node of T for all $b \in A$. Hence, $v\overline{u}b \in T$ for some v and all b, and $v\overline{u}$ is an internal node of T. It follows that $u\overline{v}$ is an internal node of $R(T)$, and so is u, and, hence, $R(T_{\mathrm{suf}}) = R(T)$. Finally, it follows from this equality and Lemma 14 that $|T_{\mathrm{suf}}| = |R(T_{\mathrm{suf}})| = |R(T)|$. □

Putting together the results of Theorem 3 and Lemma 15, and the definition of $\overline{T}$, we obtain the following theorem.

Theorem 8 *Let P be a process with minimal tree model $\langle T, p\rangle$, and let $K{=}|S_T|$. Then, the minimal tree model $\langle \widehat{T}_p, \hat{p}\rangle$ of $\overline{P}$ satisfies*

$$\sqrt{2(\alpha-1)K} - o(\sqrt{K}) \;\leq\; |S_{\widehat{T}_p}| \;\leq\; \frac{1}{2(\alpha-1)}K^2 + o(K^2)\,.$$

It follows from Theorem 8 that, when using tree data models, there might be significant differences between the size of the tree estimated by the modeler when reading the data from left to right and the size of the tree estimated when reading the data from right to left. This fact, in turn, may translate into differences in the model cost incurred by the modeler. These differences are a consequence of the choice of model class, since, as noted above, the number of free parameters determining the reversed process is identical to the number of parameters in the original process. On the other hand, it is this choice of model class that allows for efficient estimation algorithms.

6 Conclusion

`SPContextFSM` is the first algorithm for linear time encoding/decoding of a twice-universal code in the class of tree models. Beyond the practical significance of this result, in this paper we choose to emphasize and investigate the algorithmic tools employed in its derivation, as well as its information-theoretic implications. Our starting point was the identification of two basic drawbacks of full-tree models that affect the computational efficiency of the Context algorithm.

First, efficient implementations of context-based schemes require that the data collection be done in a compact suffix tree of the input sequence. Thus, the optimal full-tree model for a given sequence will generally not be a sub-tree of the suffix tree of this sequence, as it may contain paths that did not occur in the sequence and were added to make the tree full. This observation led to the GCT extension, which was investigated not only as an auxiliary data structure in a merely computational setting, but also from an information-theoretic viewpoint, as this richer model class offers potentially significant improvements in model fitting capability relative to the usual full-tree models. In this sense, we derived necessary and sufficient conditions for a GCT model to be minimal. These conditions are considerably more involved than for the sub-class of full-tree models. Consequently, the derivation of efficient algorithms for capitalizing on the potential savings offered by this model class is still an open problem of both theoretical and practical interest.

The second drawback of tree models is that they do not offer, in principle, a low-complexity mechanism for transitioning from one conditioning context to another in order to, e.g., code a sequence. We thus characterized the FSM closure of a GCT, which is instrumental in solving the context transition problem in constant time, and presented an efficient algorithm for constructing it. From an information-theoretic viewpoint, the FSM closure turned out to be closely related to the effect of time reversal on tree models, which we also investigated.

Finally, we observed that the sub-optimality of BWT-based codes in compressing tree models is the result of inverting the order in which operations are carried out in `SPContextFSM`, without any significant complexity advantage. This observation contributes to establish the standing of this transform in the universe of tree-modeling tools, where it was first placed in [22].

Appendix

A Minimal GCT models

Clearly, a model is minimal if and only if it remains so after normalization. Thus, to characterize minimality, we can assume without loss of generality that T is normal. We say that a pseudo-leaf $v \in T$ is a *pseudo-child* of $u \in T$ if $v \in \textsc{chld}_T(u)$, or v is a leaf of the form $v'b$, where $b \in A$, $v' \in \textsc{chld}_T(u)$, and $\alpha = 2$. By the discussion on normalization in Section 2-B, the case in which $v \notin \textsc{chld}_T(u)$ corresponds to the only case in which a node can be eliminated from a normal GCT in an unnormalization step making its parent a pseudo-leaf. If v or v' are atomic children of u, then v is

called an atomic pseudo-child of u. Let $L_T(u)$ denote the set of atomic pseudo-children of u, and let

$$\nu_T(u) = \max \{ 1, \ |\{ a \in A \,|\, ua \notin T \}| \} .$$

The following theorem characterizes minimal normal GCT models.

Theorem A.1 *A normal GCT model $\langle T, p \rangle$ with $A^* = A^*_P$ is minimal if and only if every node $u \in T$ satisfies the following conditions:*

(i) *Any subset of nodes in $L_T(u)$ that share a common CPMF is of size $\nu_T(u)$ or less.*

(ii) *If u is a permanent state, and v a pseudo-child of u, then $p(\cdot \,|\, u) \neq p(\cdot \,|\, v)$.*

In addition, for $\alpha = 3$, if $\langle T, p \rangle$ is minimal and $\langle T', p' \rangle$ generates the same process with T' normal, then T' is an extension of T.

It can be shown that if T is not normal, the conditions of Theorem A.1 hold on T_N if and only if they hold on T; therefore, the step of normalizing T can be avoided. The theorem implies that the minimal normal GCT model is unique for $\alpha = 3$. However, it might not be so for $\alpha \neq 3$. For example, let $\alpha = 4$, denote $A = \{a_i\}_{i=1}^4$, and consider the normal GCTs $T = \{\lambda, a_1, a_2\}$ and $T' = \{\lambda, a_3, a_4\}$. Clearly, if $p(\cdot \,|\, a_1) = p(\cdot \,|\, a_2) = p'(\cdot \,|\, \lambda)$ (as functions), $p'(\cdot \,|\, a_3) = p'(\cdot \,|\, a_4) = p(\cdot \,|\, \lambda) \neq p'(\cdot \,|\, \lambda)$, and $p(\cdot \,|\, \lambda\$) = p'(\cdot \,|\, \lambda\$)$, $\langle T, p \rangle$ and $\langle T', p' \rangle$ generate the same process and are both minimal. This example can be generalized to any $\alpha > 4$ in an obvious manner (but not to $\alpha < 4$, since T and T' must be normal). For $\alpha = 2$, let $A = \{0, 1\}$, $T = \{\lambda, 01, 10\}$, $T' = \{\lambda, 00, 11\}$, $p(\cdot \,|\, 01) = p(\cdot \,|\, 10) = p'(\cdot \,|\, \lambda)$, and $p'(\cdot \,|\, 00) = p'(\cdot \,|\, 11) = p(\cdot \,|\, \lambda) \neq p'(\cdot \,|\, \lambda)$. Again, assuming that the two models use identical CPMFs for the transient states $\lambda\$$, $0\$$, and $1\$$, both $\langle T, p \rangle$ and $\langle T', p' \rangle$ are minimal GCT models that generate the same process.

Proof. First, we show the necessity of the conditions. If u does not satisfy condition (i) we can modify the model, without affecting the process, as follows. If $u \in S_T$, add new pseudo-leaves ua for all $a \in A$ such that $ua \notin T$, and associate them with the CPMF of u. By Lemma 1, u ceases to be a permanent state. By our assumption, there are $\nu' > \nu_T(u)$ atomic pseudo-children of u that share the same CPMF. These nodes can be eliminated so that the strings they accepted are now accepted by u, which also inherits their common CPMF (for $\alpha = 2$, the elimination of an atomic pseudo-child $v \notin \text{CHLD}_T(u)$ also causes the elimination of $\text{PAR}_T(v)$, which is not a permanent state as T is normal). Overall, the number of permanent states in T decreases by at least $\nu' - \nu_T(u) > 0$. If $u \in S_T$ and does not satisfy condition (ii), then it has a pseudo-child v with the same CPMF, which can be eliminated without affecting the process, decreasing the number of permanent states of T (again, for $\alpha = 2$, the elimination of v may also imply the elimination of $\text{PAR}_T(v)$). In both cases, not satisfying the condition implies that T is not minimal. Notice that, since the CPMFs of the transient states are assumed to satisfy the constraint (3), the above state eliminations are not impeded by the transient CPMF $p(\cdot \,|\, u\$)$.

Next, we prove the sufficiency of the conditions. Consider two normal GCT models $\langle T, p \rangle$ and $\langle T', p' \rangle$ that generate the same process P, with $A^* = A^*_P$, and assume that $\langle T, p \rangle$ satisfies conditions (i) and (ii). We first state two general properties that will be used in the proof.

(P1) If $u \in A^*$ is such that $u \notin \text{WORD}(T')$, then $V_T(u) \in S_T$, $V_{T'}(u) \in S_{T'}$, and $p(\cdot|\, V_T(u)) = p'(\cdot|\, V_{T'}(u))$ (as functions). Moreover, if $u \in T$, then u is a leaf of T.

(P2) If $u \in T \setminus T'$, then either u is a pseudo-leaf of T, or $\alpha = 2$ and there exists $b \in A$ such that ub is a leaf of T. In addition, denoting by v the claimed pseudo-child of $\text{PAR}_T(u)$ (either u or ub), we have $p(\cdot|\, v) = p'(\cdot|\, V_{T'}(u))$ (as functions).

For $u \notin T$, **(P1)** is an obvious consequence of the existence of $b \in A$ such that $ub \notin \text{WORD}(T)$, of Lemma 1, and of $P(\cdot|z\overline{u})$ being independent of $z \in A^*$ (as the two processes are identical and all strings have nonzero probability); conditions (i) and (ii) on $\langle T, p\rangle$ are not required. The case $u \in T$ and the second part of **(P1)** follow from the fact that any nontrivial subtree of T must contain permanent states with at least two different CPMFs, for otherwise the subtree would contain a node all of whose children are leaves and that violates either condition (i) or condition (ii).

To prove **(P2)**, observe first that since $u \notin T'$ the set $A_u = \{a \in A : ua \notin \text{WORD}(T')\}$ contains at least $\alpha - 1$ symbols. Thus, since T is normal, if $u \in S_T$ then there exists a symbol $a' \in A_u$ such that $ua' \notin T$, and therefore, by **(P1)**, $p(\cdot|\, u) = p'(\cdot|\, V_{T'}(u))$. Now, consider the strings $ub \in \text{WORD}(T)$, $b \in A_u$, and let $uby \in \text{CHLD}_T(u)$, $y \in A^*$. By **(P1)**, uby must be a leaf of T. If u is not a pseudo-leaf of T, then there exists at least one such leaf uby and, moreover, uc must be a leaf of T for *every* $c \in A_u$, for otherwise $u \in S_T$ and it would have the same CPMF $p'(\cdot|\, V_{T'}(u))$ as uby, violating condition (ii). This case can only occur for $\alpha = 2$, for otherwise the number of leaves sharing the same CPMF would be $\alpha - 1 > 1 = \nu_T(u)$, violating condition (i). The proof of **(P2)** is complete.

Now, to prove the sufficiency of conditions (i) and (ii), we will show that if a GCT model $\langle T', p'\rangle$ generates the same process as $\langle T, p\rangle$ and it also satisfies the conditions, then T and T' are identical up to transformations of $\langle T', p'\rangle$ that do not affect neither $|S_{T'}|$ nor the generated process. Without loss of generality, we can assume that T' is also normal. Clearly, it suffices to prove that if $u \in T \cap T'$, then after such transformations, u is unaffected and $\text{CHLD}_T(u) = \text{CHLD}_{T'}(u)$.

The claim is obvious for $u \notin S_T \cup S_{T'}$, as u has a full complement of atomic children in both GCTs. Next, we show that if $u \in S_T$, then $u \in S_{T'}$ or, equivalently, that $u \in S_T \setminus S_{T'}$ implies $u \notin T'$. If the claim did not hold, we would have $uc \in T'$ for all $c \in A$. Since T is normal and $u \in S_T$, there exist $a, a' \in A$, $a \neq a'$, such that $ua, ua' \notin T$. Thus, by **(P2)**, T' has pseudo-children in the directions of a and a' sharing the same CPMF. Since $\nu_{T'}(u) = 1$, T' does not satisfy condition (i), a contradiction.

Consequently, it suffices to consider the case $u \in S_T \cap S_{T'}$. Assume first $\alpha > 2$. If $ua \in T \setminus T'$ for some $a \in A$, then, by **(P2)**, ua is a pseudo-leaf of T and $p'(\cdot|\, u) = p(\cdot|\, ua)$. Therefore, if $p(\cdot|\, u) = p'(\cdot|\, u)$, the sets of atomic children of u for T and T' coincide for otherwise condition (ii) is violated. If $p(\cdot|\, u) \neq p'(\cdot|\, u)$, we must have $ua \in T \cup T'$ for all $a \in A$, for otherwise there exists $b \in A$ such that $uab \notin \text{WORD}(T')$ and $uab \notin T$, and a contradiction to **(P1)** follows. Moreover, in order for both T and T' to satisfy condition (i), the sets $\{ua \in T \setminus T'\}$ and $\{ua \in T' \setminus T\}$ must have the same size. Therefore, by deleting from T' all nodes in the latter set (which are pseudo-leaves) and adding the nodes in the former set, the size of $S_{T'}$ remains unchanged, while the process $\langle T', p'\rangle$ is preserved by replacing the distribution $p'(\cdot|\, u)$ with $p(\cdot|\, u)$ and associating $p'(\cdot|\, u)$ with the added pseudo-leaves (this operation does not affect the transient states). After this transformation, again, the sets of atomic children of T and T' coincide, and $p(\cdot|\, u) = p'(\cdot|\, u)$.

For $\alpha = 2$, the two sets of atomic children are empty by normality. We show that we can also assume $p(\cdot|\, u) = p'(\cdot|\, u)$. For all $c \in A$, there exists $b \in A$ such that $ucb \notin \text{WORD}(T')$. If $p(\cdot|\, u) \neq p'(\cdot|\, u)$, then ucb is a leaf of T, for otherwise a contradiction to **(P1)** would follow. We can assume, without loss of generality, that $c = b$. Thus, letting $A = \{a, a'\}$, T has leaves uaa and $ua'a'$, with CPMF $p'(\cdot|\, u)$. Similarly, T' has leaves uaa' and $ua'a$, with CPMF $p(\cdot|\, u)$. Therefore, we can replace the subtree of T' rooted at u with the corresponding subtree of T (replacing also the associated CPMFs), without affecting the size of $S_{T'}$ or the generated process.

To complete the sufficiency proof, it remains to show that, for any α, if $uay \in \text{CHLD}_T(u)$ and $uaz \in \text{CHLD}_{T'}(u)$ for some $a \in A$, $y, z \in A^+$, and $p(\cdot|\, u) = p'(\cdot|\, u)$, then $y = z$. Suppose it is not. Then, either $V_{T'}(uay) = u$ or $V_T(uaz) = u$. Assume, without loss of generality, that the former holds. Then, by **(P2)**, u has a pseudo-child v with $p(\cdot|\, v) = p'(\cdot|\, u) = p(\cdot|\, u)$, violating condition (ii).

Finally, assume that $\langle T', p'\rangle$ generates the same process as $\langle T, p\rangle$, with T' normal, $\langle T, p\rangle$ minimal, and $\alpha = 3$. We prove that $T \subseteq T'$. Since $T \subseteq T_N$, we can assume, without loss of generality, that T is normal. Suppose, to the contrary, that $w \in T \setminus T'$. Clearly, there exists $v \preceq w$ such that $v \in T \setminus T'$ and $\text{PAR}_T(v) \preceq V_{T'}(v) \stackrel{\triangle}{=} u$. Let $v = uax$, with $a \in A$ and $x \in A^*$. By **(P2)**, v is a pseudo-leaf of T and $p(\cdot|\, v) = p'(\cdot|\, u)$. Since T' is normal, there exists $b \in A \setminus \{a\}$ such that $ub \notin T'$. Moreover, for any $a' \in \{a, b\}$ we must have $ua' \in T$, for otherwise $V_T(u) \in S_T$ and $ua'c \notin T \cup \text{WORD}(T')$ for some $c \in A$, implying, by **(P1)**, $p(\cdot|\, V_T(u)) = p'(\cdot|\, u) = p(\cdot|\, v)$, violating condition (ii) for T. Thus, $u \in T$ and $v = ua$. Again by **(P2)**, ub is a pseudo-leaf of T with $p'(\cdot|\, u) = p(\cdot|\, ub)$. It follows that $\{ua, ub\}$ is a subset of pseudo-leaves in $L_T(u)$ that share the same CPMF, violating condition (i) for T since $\nu_T(u) = 1$ as $\alpha = 3$. □

B Proof of Theorem 4

We claim that the total number of comparisons made during computations of $C_T(x)$ is upper-bounded by $2N_E + N'$. By the preceding discussion, this fact establishes the claimed upper bound, as the other operations take constant time per node of T_{suf}.

Let $T_0 = T$, and, for $i > 0$, let T_i be a snapshot of T' after the i-th computation of $C_T(x)$ in Step 2 of `Verify`, and the corresponding call to `Insert` in Step 4, if such a call was made. Let C_i be the total number of comparisons made in computations of $C_T(\cdot)$ up to that point, and C the total number after completion of the algorithm, when $T' = T_{\text{suf}}$. Denote by T_i^* the set of nodes of T_i excluding the root, by T_i^- the subset of those nodes that have not been visited at the time of the snapshot, and by $T_{L,i}^-$ the subset of nodes in T_i^- that are leaves of T_i. We construct two sequences of functions $f_i : T_i^* \to \mathbb{Z}$ and $g_i : T_i^* \to \mathbb{Z}$, $i \geq 0$, such that

$$f_0(uv) = 2|v|, \quad uv \in T_0^*, \;\; u = \text{PAR}_{T_0^*}(uv), \;\; u \xrightarrow{v} uv,$$

$g_0 \equiv 0$, and, for $i > 0$, and each node $uav \in T_0^*$, with $u = \text{PAR}_{T_0^*}(uav)$,

$$f_i(uav) = \begin{cases} 2|av| & uav \in T_{L,i}^- \text{ or } \texttt{Traversed[}u, a\texttt{]=}\textit{false}, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$g_i(uav) = \begin{cases} |av| & uav \in T_i^- \setminus T_{L,i}^- \text{ and } \texttt{Traversed[}u, a\texttt{]=}\textit{true}, \\ 0 & \text{otherwise.} \end{cases}$$

Notice that the condition for $g_i(uav) \neq 0$ can only hold when uav was just created as a result of an edge split in Step 5 of `Insert`. We prove, by induction on i, that the following condition holds for all $i \geq 0$ for which the relevant quantities are defined:

$$C_i \leq |T_i^*| + 2N_E - \sum_{t \in T_i^*} f_i(t) - \sum_{t \in T_i^*} g_i(t). \tag{B.1}$$

Condition (B.1) implies that at the end of the execution of the algorithm, after all nodes have been visited, we have $C \leq N' + 2N_E$, as claimed. The inequality clearly holds for $i = 0$. Assume now it holds for all $i \leq n-1$. We determine f_n and g_n after the next execution of Step 2 and any necessary insertions, assuming `Verify` was called with argument $w = cx$, and $C_T(x) = \langle r, u, v \rangle$. We have $f_n(cx) = g_n(cx) = 0$, since cx is being visited. If $u \neq \lambda$, let $u = au'$, $a \in A$. In this case, `Insert` was called, and an internal node was created by splitting an edge $r \xrightarrow{uy} ruy$ into $r \xrightarrow{u} ru \xrightarrow{y} ruy$. Hence, we have

$$f_n(ru) = \begin{cases} 2|u| & \texttt{Traversed[}r, a\texttt{]} = \textit{false}, \\ 0 & \text{otherwise,} \end{cases} \tag{B.2}$$

$$f_n(ruy) = \begin{cases} 2|y| & \texttt{Traversed[}r, a\texttt{]} = \textit{false}, \\ 0 & \text{otherwise,} \end{cases} \tag{B.3}$$

and

$$g_n(ru) = \begin{cases} |u| & \texttt{Traversed[}r, a\texttt{]} = \textit{true}, \\ 0 & \text{otherwise.} \end{cases} \tag{B.4}$$

For node ruy, notice that if `Traversed[`r, a`]` $= \textit{true}$, ruy had been visited either immediately after creation from Step 7 or 9, or after setting `Traversed(`r, a`)` $= \textit{true}$ from Step 16. Therefore, we have $g_n(ruy) = 0$. If $v \neq \lambda$, a new node ruv was created, and has not yet been visited, i.e. $ruv \in T_{L,i}^-$. Thus, we have $f_n(ruv) = 2|v|$, and $g_n(ruv) = 0$. All other values of f_n and g_n are unchanged from their values at $i = n-1$. We prove that the condition in (B.1) holds for $i = n$. Assume first that the invocation of `Verify` at which snapshot n was taken was made recursively from Step 7. It follows from the discussion preceding the theorem that in this case, $C_{T'}$ was computed in *fast* mode, since we know that x was a word of T_{n-1}. This also implies that $v = \lambda$, and ru was the only node possibly created (if any). Now, if the invocation of which Step 7 was part was with argument $\hat{c}\hat{x}$, and $C_{T'}(\hat{x}) = \langle \hat{r}, \hat{u}, \hat{v} \rangle$, then the number of comparisons needed in the fast computation of $C_{T'}(x)$ is upper bounded by $|\hat{u}| - |u| + 1$ if $u \neq \lambda$, or $|\hat{u}|$ otherwise. Thus, we can write

$$C_n - C_{n-1} \leq |\hat{u}| - |u| + |T_n^*| - |T_{n-1}^*|. \tag{B.5}$$

Also, we have $f_n(cx) = 0$, and $f_{n-1}(cx) = 0$, since we call `Verify` from Step 7 only for nodes that are not leaves, and whose incoming edge has `Traversed` $= \textit{true}$. Now, it follows from (B.2), (B.3), and the fact that edges resulting from a split inherit the "traversed" status of the original edge, that

$$f_n(ru) + f_n(ruy) - f_{n-1}(ruy) = 0. \tag{B.6}$$

Therefore, we have

$$\sum_{t \in T_n^*} f_n(t) - \sum_{t \in T_{n-1}^*} f_{n-1}(t) = 0. \tag{B.7}$$

As for the functions g, we have $g_n(cx) = 0$,

$$g_{n-1}(cx) = |\hat{u}|, \tag{B.8}$$

since coming from Step 7, cx is a newly created internal node that did not exist at $i = n-1$,

$$g_n(ru) \leq |u|, \tag{B.9}$$

$$g_n(ruy) = 0, \tag{B.10}$$

and

$$g_{n-1}(ruy) = 0 \text{ whenever } ruy \neq cx. \tag{B.11}$$

The last two equations follow from the fact that if the incoming edge of ruy has `Traversed`$=$ *true*, the node had already been visited at $i = n-1$. The only exception is the coincidental case where $ruy = cx$, the node being visited at $i = n$, in which case the node had not been visited at $i = n-1$ and the value from (B.8) takes precedence. Other values of g_n remain unchanged from g_{n-1}, and, hence, it follows from (B.8)–(B.11) that

$$-\sum_{t \in T_n^*} g_n(t) + \sum_{t \in T_{n-1}^*} g_{n-1}(t) \geq |\hat{u}| - |u|. \tag{B.12}$$

Now, from (B.5),(B.7),(B.12), and the induction hypothesis we obtain (B.1) for $i = n$, as desired.

It remains to consider the case where the invocation of `Verify` at which snapshot n was taken was not made from Step 7. In this case, the number of comparisons made in computing $C_{T'}(x)$ is $|\hat{u}|-|v|+1$ when $v \neq \lambda$, or $|\hat{u}|$ otherwise, where $\langle r, u, v\rangle$ and $\langle \hat{r}, \hat{u}, \hat{v}\rangle$ are defined as before. Thus,

$$C_n - C_{n-1} \leq |\hat{u}| - |v| + |T_n^*| - |T_{n-1}^*|. \tag{B.13}$$

Using reasoning very similar to the previous case, we also obtain

$$f_n(cx) = 0, \quad f_{n-1}(cx) = 2|\hat{u}|, \quad f_n(ru) + f_n(ruy) - f_{n-1}(ruy) = 0,$$

and

$$-\sum_{t \in T_n^*} f_n(v) + \sum_{t \in T_{n-1}^*} f_{n-1}(t) = 2|\hat{u}| - 2|v|. \tag{B.14}$$

Also,

$$g_n(cx) = 0, \quad g_{n-1}(cx) = 0,$$

$$g_n(ru) \leq |u|, \quad g_n(ruv) = 0, \quad g_n(ruy) = 0$$

$$g_{n-1}(ruy) = 0 \text{ whenever } ruy \neq cx,$$

and, thus,

$$-\sum_{t \in T_n^*} g_n(t) + \sum_{t \in T_{n-1}^*} g_{n-1}(t) \geq -|u|. \tag{B.15}$$

From (B.14) and (B.15) we obtain

$$-\sum_{t\in T_n^*} f_n(t) + \sum_{t\in T_{n-1}^*} f_{n-1}(t) - \sum_{t\in T_n^*} g_n(t) + \sum_{t\in T_{n-1}^*} g_{n-1}(t) \geq 2|\hat{u}| - 2|v| - |u| \geq |\hat{u}| - |v|, \tag{B.16}$$

where the rightmost inequality follows from $|\hat{u}| \geq |u|+|v|$, which in turn follows from $ruv = x = \text{tail}(\hat{r})\hat{u}$ and $\text{tail}(\hat{r}) \preceq r$. Finally, combining with (B.13), and applying the induction hypothesis, we obtain the desired result. □

C Linear time decoding

The relation between $\hat{s}_i \stackrel{\triangle}{=} s_{\hat{T}'_F(x)}(x^i)$ and $s_i \stackrel{\triangle}{=} s_{T_F(x)}(x^i)$ is given by Lemma C.1 below, for which we remove the \$ symbols from transient states, and define z_i such that $\hat{s}_i z_i$ is the longest prefix of $\overline{x}^i$ in $\text{WORD}(\hat{T}'_F(x))$. Further, define $b_i \stackrel{\triangle}{=} x_{i-|\hat{s}_i z_i|}$ in case $|\hat{s}_i z_i| < i$, or $b_i = \lambda$ otherwise (these definitions are illustrated in Figure 8 of Section 4).

Lemma C.1 *For every i, $0 \leq i < n$, we have $s_i = \hat{s}_i z_i b_i$.*

Proof. Since $T_F(x)$ is a full tree, it suffices to show that $\hat{s}_i z_i b_i \in \text{WORD}(T_F(x))$, and that either $b_i = \lambda$, or $\hat{s}_i z_i b_i c \notin \text{WORD}(T_F(x))$ for any $c \in A$. To prove the first claim, observe that, by definition, $\hat{s}_i z_i \in \text{WORD}(\hat{T}'_F(x))$, so that there exists a string y such that $\hat{s}_i z_i y \in \hat{T}'_F(x)$. Thus, by construction of the FSM closure, there exists another string u for which $u\hat{s}_i z_i y \in \hat{T}'(x)$, implying that $u\hat{s}_i z_i y b \in T(x)$ for every $b \in A$. Consequently, $\hat{s}_i z_i y b \in T_F(x)$, and the claim follows from the fact that $T_F(x)$ is full. As for the second claim, assume that $\hat{s}_i z_i b_i c \in \text{WORD}(T_F(x))$ for some $c \in A$. Since $T_F(x)$ is full, this assumption implies that $\hat{s}_i z_i b_i c \in T_F(x)$ for all $c \in A$, so that there exists a string v for which $v\hat{s}_i z_i b_i c \in T(x)$. Thus, $v\hat{s}_i z_i b_i \in \text{WORD}(\hat{T}'(x))$, implying that $v\hat{s}_i z_i b_i w \in \hat{T}'(x)$ for some string w, and further that $\hat{s}_i z_i b_i w \in \hat{T}'_F(x)$. Since, by definition, $\hat{s}_i z_i$ is the longest prefix of $\overline{x}^i$ that is a word of $\hat{T}'_F(x)$, we must then have $b_i = \lambda$. □

Next, we show that, in fact, it is not necessary to revisit the decoded sequence for all the symbols in z_i in order to determine s_i. Observe that, by the FSM property of $\hat{T}'_F(x)$, $\hat{s}_{i+1} \preceq x_{i+1}\hat{s}_i$. Removing again the \$ symbols from transient states, define u_{i+1} to be the string satisfying $x_{i+1}\hat{s}_i = \hat{s}_{i+1}u_{i+1}$.

Lemma C.2 *If $\hat{s}_{i+1}u_{i+1}\text{head}(z_i) \in \text{WORD}(\hat{T}'_F(x))$, then $z_{i+1} = u_{i+1}z_i$.*

Proof. By the definition of z_i and u_i, there exist sequences t and v such that

$$\overline{x}^{i+1} = x_{i+1}\hat{s}_i z_i b_i t = \hat{s}_{i+1} z_{i+1} v = \hat{s}_{i+1} u_{i+1} z_i b_i t\,. \tag{C.1}$$

Further, by the assumption of the lemma and the definition of z_{i+1}, we have $u_{i+1}\text{head}(z_i) \preceq z_{i+1}$; we show that we also have $z_{i+1} \preceq u_{i+1}z_i$. Otherwise, by (C.1) and the definition of z_{i+1}, $\hat{s}_{i+1}u_{i+1}z_i b_i$ would be a word of $\hat{T}'_F(x)$ with $b_i \neq \lambda$, implying the existence of a string y such that $\hat{s}_{i+1}u_{i+1}z_i b_i y \in \hat{T}'_F(x)$. Thus, since $\hat{T}'_F(x)$ is FSM, $\hat{s}_i z_i b_i y$ is also a node by (C.1), so that $\hat{s}_i z_i b_i \in \text{WORD}(\hat{T}'_F(x))$, in contradiction with the definition of z_i. Now, if $z_i = \lambda$, the proof is complete. If $z_i \neq \lambda$, we then have $z_{i+1} = u_{i+1}z'_i$,

where z'_i is a non-empty prefix of z_i, and define z''_i by $z_i = z'_i z''_i$. The proof is complete if we show that $z''_i = \lambda$. There exists a string w such that $\hat{s}_{i+1}u_{i+1}z'_i w$ is a node of $\hat{T}'_F(x)$ and, by the FSM property, so is $\hat{s}_i z'_i w$. Moreover, $w \neq \lambda$, for otherwise $\hat{s}_i z'_i$ would be a node, and thus $\hat{s}_i$ would not be the state at time i. If $z''_i \neq \lambda$, we have $\text{head}(w) \neq \text{head}(z''_i)$, for otherwise $\hat{s}_{i+1}z_{i+1}\text{head}(z''_i) \in \text{WORD}(\hat{T}'_F(x))$, contradicting the definition of z_{i+1}. It follows that $\hat{s}_i z'_i \text{head}(w)$ and $\hat{s}_i z'_i \text{head}(z''_i)$ are two different words in $\hat{T}'_F(x)$, making $\hat{s}_i z'_i$ a node, a contradiction. □

Given $\hat{s}_i$, $|u_i|$, and $|z_{i-1}|$ (starting with $z_0 = \lambda$), we can recursively determine $|z_i|$ by checking decoded symbols and descending $\hat{T}'_F(x)$, starting from $\hat{s}_i$, in the direction $x_{i-|\hat{s}_i|}, x_{i-|\hat{s}_i|-1}, \cdots, x_{i-|\hat{s}_i|-|u_i|-1}$. If, at some point, the concatenated string is not a word of $\hat{T}'_F(x)$, by definition, we have determined $|z_i|$. Otherwise, $\hat{s}_i u_i \text{head}(z_{i-1}) \in \text{WORD}(\hat{T}'_F(x))$, and, by Lemma C.2, $|z_i| = |u_i| + |z_{i-1}|$. Thus, the determination of $|z_i|$ requires at most $|u_i| + 1$ comparisons. Since $|u_i| = |\hat{s}_{i-1}| - |\hat{s}_i| + 1$, we need at most n comparisons along x^n.

Now, given $\hat{s}_i$, $|z_i|$, $\text{head}(z_i)$, and b_i, it is easy to determine $s_{T(x)}(x^i)$ (which contains the decoding statistics) in constant time per input symbol by defining an additional data structure. Specifically, for every internal node u of $T(x)$ that is also a node of $\hat{T}'_F(x)$, consider the set A_u of symbols for which u has an edge of $\hat{T}'(x)$ in their direction, and let $u(a)$ denote the edge of $\hat{T}'_F(x)$ in the direction of $a \in A_u$. For every j, $1 \leq j < |u(a)|$, let $v_{u,a}(j)$ denote the node of $T(x)$ obtained by concatenating j symbols of $u(a)$ to u. A data structure `Jump[`u`]`, linking u with each $v_{u,a}(j)$, can be built in constant time for all relevant nodes, e.g., by initially setting up the data structure for the nodes of $\hat{T}'(x)$, and then updating it as edges of $\hat{T}'(x)$ are split by `MakeFSM`. In addition, for a node w of $\hat{T}'_F(x)$ that is not a node of $T(x)$, the initialization of `Origin[`w`]` in `MakeFSM` can readily be modified so that it points to its ancestor in $T(x)$, rather to an ancestor in $\hat{T}'(x)$. Equipped with the data structures `Jump[`u`]`, and based on Lemmas C.1 and C.2, we can determine $s_{T(x)}(x^i)$ for $z_i \neq \lambda$ as follows:

- If $\hat{s}_i$ is an internal node of $T(x)$ and $\text{head}(z_i) \in A_{\hat{s}_i}$, then $\hat{s}_i z_i$ is given by $v_{\hat{s}_i,\text{head}(z_i)}(|z_i|)$ (by definition, $|z_i| < |\hat{s}_i(\text{head}(z_i))|$), and $s_{T(x)}(x^i) = s_i = \hat{s}_i z_i b_i$ (Lemma C.1).
- If $\hat{s}_i$ is an internal node of $T(x)$ and $\text{head}(z_i) \notin A_{\hat{s}_i}$, then $s_{T(x)}(x^i) = \hat{s}_i \text{head}(z_i)$.
- If $\hat{s}_i$ is a leaf of $T(x)$, then $s_{T(x)}(x^i) = \hat{s}_i$.
- If $\hat{s}_i$ is not a node of $T(x)$, then `Origin[`$\hat{s}_i$`]` points to $s_{T(x)}(x^i)$.

When $z_i = \lambda$, $s_{T(x)}(x^i) = \hat{s}_i$ if $\hat{s}_i$ is a node of $T(x)$, and `Origin[`$\hat{s}_i$`]` points to $s_{T(x)}(x^i)$ otherwise.

References

[1] J. Rissanen, "Universal coding, information, prediction, and estimation," *IEEE Trans. Inform. Theory*, vol. IT-30, pp. 629–636, July 1984.

[2] J. Rissanen, "A universal data compression system," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 656–664, Sept. 1983.

[3] M. J. Weinberger, J. Rissanen, and M. Feder, "A universal finite memory source," *IEEE Trans. Inform. Theory*, vol. IT-41, pp. 643–652, May 1995.

[4] P. L. Buhlmann and A. Wyner, "Variable length Markov chains," *Annals of Statistics*, vol. 27, pp. 480–513, 1998.

[5] R. Nohre, *Some Topics in Descriptive Complexity.* PhD thesis, Department of Computer Science, The Technical University of Linkoping, Sweden, 1994.

[6] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "The context-tree weighting method: Basic properties," *IEEE Trans. Inform. Theory*, vol. IT-41, pp. 653–664, May 1995.

[7] F. M. J. Willems, "The context-tree weighting method: Extensions," *IEEE Trans. Inform. Theory*, vol. IT-44, pp. 792–798, Mar. 1998.

[8] B. Ryabko, "Prediction of random sequences and universal coding," *Problems of Information Transmission*, vol. 24, pp. 87–96, April/June 1988.

[9] M. J. Weinberger, A. Lempel, and J. Ziv, "A sequential algorithm for the universal coding of finite-memory sources," *IEEE Trans. Inform. Theory*, vol. IT-38, pp. 1002–1014, May 1992.

[10] B. Ryabko, "Twice-universal coding," *Problems of Information Transmission*, vol. 20, pp. 173–177, July/September 1984.

[11] N. Merhav and M. Feder, "Universal prediction," *IEEE Trans. Inform. Theory*, vol. IT-44, pp. 2124–2147, Oct. 1998.

[12] M. J. Weinberger, N. Merhav, and M. Feder, "Optimal sequential probability assignment for individual sequences," *IEEE Trans. Inform. Theory*, vol. IT-40, pp. 384–396, Mar. 1994.

[13] J. Rissanen, "Stochastic complexity and modeling," *Annals of Statistics*, vol. 14, pp. 1080–1100, Sept. 1986.

[14] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "Context-tree maximizing," in *Proc. 2000 Conference on Information Sciences and Systems*, (Princeton, New Jersey, USA), pp. TP6–7–TP6–12, Mar. 2000.

[15] R. E. Krichevskii and V. K. Trofimov, "The performance of universal encoding," *IEEE Trans. Inform. Theory*, vol. IT-27, pp. 199–207, Mar 1981.

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

[17] D. Baron and Y. Bresler, "An $O(n)$ semi-predictive universal encoder via the BWT." To appear in *IEEE Trans. Inform. Theory.*

[18] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. 32 (4), pp. 396–402, Apr. 1984.

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

[20] M. Effros, "PPM performance with BWT complexity: A fast and effective data compression algorithm," *Proceedings of the IEEE*, vol. 88, pp. 1703–1712, Nov. 2000.

[21] N. J. Larsson, "Extended application of suffix trees to data compression," in *Proc. 1996 Data Compression Conference*, (Snowbird, Utah, USA), pp. 190–199, Apr. 1996.

[22] M. Effros, K. Visweswariah, S. Kulkarni, and S. Verdú, "Universal lossless source coding with the Burrows-Wheeler transform," *IEEE Trans. Inform. Theory*, vol. IT-48, pp. 1061–1081, May 2002.

[23] N. J. Larsson, "The context trees of block sorting compression," in *Proc. 1998 Data Compression Conference*, (Snowbird, Utah, USA), pp. 189–198, Mar. 1998.

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

[25] J. Rissanen and G. G. Langdon, "Universal modeling and coding," *IEEE Trans. Inform. Theory*, vol. IT-27, pp. 12–23, Jan. 1981.

[26] R. B. Ash, *Information Theory.* John Wiley & Sons, Inc., 1967.

[27] W. Feller, *Probability theory and its applications*, vol. 1. New York: John Wiley & Sons, Inc., third ed., 1968.

[28] D. E. Knuth, *The Art of Computer Programming. Fundamental Algorithms*, vol. 1. Reading, MA: Addison-Wesley, third ed., 1997.

[29] D. E. Knuth, *The Art of Computer Programming. Sorting and Searching*, vol. 3. Reading, MA: Addison-Wesley, second ed., 1997.

[30] D. R. Morrison, "Patricia - practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[31] W. Szpankowski, *Average Case Analysis of Algorithms on Sequences.* New York: John Wiley & Sons, Inc., 2001.

[32] P. Weiner, "Linear pattern matching algorithms," in *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[33] E. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, no. 2, pp. 262–272, 1976.

[34] M. Feder, N. Merhav, and M. Gutman, "Universal prediction of individual sequences," *IEEE Trans. Inform. Theory*, vol. IT-38, pp. 1258–1270, July 1992.

[35] M. J. Weinberger and M. Feder, "Predictive stochastic complexity and model estimation for finite-state processes," *Journal of Statistical Planning and Inference*, vol. 39, pp. 353–372, 1994.

[36] N. G. de Bruijn, "A combinatorial problem," *Koninklijke Nederlands Akademie van Wetenschappen, Proceedings*, vol. 49 Part 2, pp. 758–764, 1946.

[37] S. W. Golomb, *Shift Register Sequences.* San Francisco: Holden-Day, 1967.

[38] J. Marshall Hall, *Combinatorial Theory.* New York: John Wiley & Sons, second ed., 1986.

[39] A. Martín, G. Seroussi, and M. J. Weinberger, "Linear time universal coding of tree sources." Hewlett-Packard Laboratories Technical Report, May 2003.

[40] T. M. Cover, "Enumerative source encoding," *IEEE Trans. Inform. Theory*, vol. IT-19, pp. 73–77, Jan. 1973.

[41] D. Preuss, "Two-dimensional facsimile source encoding based on a Markov model," *Nachrichtentechn. Zeitschrift*, vol. 28, pp. 358–363, Oct. 1975.

[42] D. Gillman and R. L. Rivest, "Complete variable-length "fix-free" codes," *Designs, Codes and Cryptography*, vol. 5, no. 2, pp. 109–114, 1995.