

1999

# On Learning Probabilistic Automata

Alberto Apostolico

Report Number:  
99-039

---

Apostolico, Alberto, "On Learning Probabilistic Automata" (1999). *Computer Science Technical Reports*. Paper 1469.  
<http://docs.lib.purdue.edu/cstech/1469>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

**ON LEARNING PROBABILISTIC AUTOMATA**

**Alberto Apostolico**

**Department of Computer Sciences  
Purdue University  
West Lafayette, IN 47907**

**CSD TR #99-039  
November 1999**

# ON LEARNING PROBABILISTIC AUTOMATA\*

Alberto Apostolico<sup>†</sup>

Purdue University and Università di Padova

## 1 Introduction and Summary

Probabilistic models of various classes of sources are developed in the context of coding and compression as well as in machine learning and classification. In the first domain, the repetitive structures of substrings are regarded as redundancies and sought to be removed. In the second, repeated subpatterns are unveiled as carriers of information and structure. Source modeling is made hard in practice by the fact that we do not know the source probabilities, the latter being actually rather fictitious entities or models. In fact, one rather pervasive problem is precisely that of learning or estimating these probabilities from the observed strings. This problem is twofold, and its two components are tightly coupled in practice. From an information theoretic standpoint, the question is how to define notions of probability and information relative to a class of sources. Once one such characterization is agreed upon, interesting algorithmic questions may revolve around the computational cost inherent to the process of learning or estimating probabilities within the given class (see, e.g., [1, 5, 6, 13, 15] and references therein).

A popular approach to the statistical modeling of sequences relies on the structure of uniform, fixed-memory Markov models (we refer to, e.g., [11, 13, 14] in the context of predictive and universal codes, [7, 15] in the context of learning and classification). For sequences in important families, such as those arising in applications that range from natural language, to speech, handwriting, and molecular sequence analysis, the autocorrelation or “memory” exhibited decays exponentially fast with length. In other words, there is a maximum length  $L$  of the recent history of a sequence, above which the empirical probability distribution of next symbol given the the last  $L' > L$  symbols does not change appreciably. It is possible and customary

---

\*Work supported in part by NSF Grant CCR-9700276 and by the Italian Ministry of Research.

<sup>†</sup>Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA and Dipartimento di Elettronica e Informatica, Università di Padova, Via Gradenigo 6/A, I-35131 Padova, Italy. axa@cs.purdue.edu.

to model these sources by Markov chains of order  $L$ , this maximum useful memory length. Even so, such automata tend to be in practice unnecessarily bulky and computationally imposing both during their synthesis and use. In [15], much more compact, tree-shaped variants of probabilistic automata are built which assume an underlying Markov process of variable memory length not exceeding some maximum  $L$ . The probability distributions generated by these automata is equivalent to that of a Markov chain of order  $L$ , but the description of the automaton itself is much more succinct. The process of learning the automaton from a given training set  $S$  of sequences requires  $\Theta(Ln^2)$  worst-case time, where  $n$  is the total length of the sequences in  $S$  and  $L$  is the length of a longest substring of  $S$  to be considered for a candidate state in the automaton. Once the automaton is built, predicting the likelihood of a query sequence of  $m$  characters may cost time  $\Theta(m^2)$  in the worst case.

Here, we present automata equivalent to PSTs but having the property that learning the automaton takes  $O(n)$  time. Along the way, we address notions of empirical probability and their efficient computation, possibly a by-product of more general interest.

## 2 Learning Variable-Memory PSAs

We deal with a (possibly singleton) collection  $S$  of strings over a finite alphabet  $\Sigma$ , and use  $\lambda$  to denote the empty string. The *length* of  $S$  is the sum of the lengths of all strings in it and is denoted by  $n$ . With reference to  $S$  and a generic string  $s = s_1s_2\dots s_l$ , the *empirical probability*  $\tilde{P}$  of  $s$  is defined provisionally as the number of times  $s$  occurs in  $S$  divided by the maximum “possible” number of such occurrences. The *conditional empirical probability* of observing the symbol  $\sigma$  soon after the string  $s$  is given by the ratio

$$\tilde{P}(\sigma|s) = \frac{\chi_{s\sigma}}{\chi_{s*}},$$

where  $\chi_w$  is the number of occurrences of string  $w$  and  $s*$  is every single-symbol extension of  $s$  having occurrence in  $S$ . Finally,  $\text{suf}(s)$  denotes  $s_2s_3\dots s_l$ .

We recall the structure of the tree-shaped probabilistic automata as described in [7, 15], where they were used as an aid in the classification of strings. In any such tree, each edge is labeled by a symbol, each node corresponds to a unique string—the one obtained by traveling from that node to the root—and nodes are weighted by a probability vector giving the distribution over the next symbol. In the following,  $\mathcal{T}$  is the tree-shaped automaton,  $\bar{S}$  is the set of strings that we want to check and  $\gamma_s$  is the probability distribution over the next symbol associated with node  $s$ . The construction starts with a tree consisting of just the root node (i.e., the tree associated with the empty word) and adds paths as follows. For each substring  $s$  considered, it is checked whether there is some symbol  $\sigma$  in the alphabet for which the empirical probability of observing it after  $s$  is significant and significantly different from the probability of observing it after  $\text{suf}(s)$ . Whenever these conditions hold, the path relative to the substring (and possibly its necessary but currently missing ancestors) are added to the tree.

1. Initialize  $\bar{T}$  and  $\bar{S}$ : let  $\bar{T}$  consist of a single node corresponding to  $e$ , and let  $\bar{S} \leftarrow \{\sigma | \sigma \in \Sigma \text{ and } \bar{P}(\sigma) \geq P_{min}\}$ .
2. Building the Skeleton: While  $\bar{S} \neq 0$ , pick any  $s \in \bar{S}$  and do
  - (A) Remove  $s$  from  $\bar{S}$ ;
  - (B) if there is a symbol  $\sigma \in \Sigma$  such that

$$\bar{P}(\sigma|s) \geq (1 + \alpha)\gamma_{min},$$

and

$$(1) \frac{\tilde{P}(\sigma|s)}{\bar{P}(\sigma|suffix(s))} \geq r \text{ or } (2) \frac{\tilde{P}(\sigma|s)}{\bar{P}(\sigma|suffix(s))} \leq 1/r,$$

then add to  $\bar{T}$  the node corresponding to  $s$  and all the nodes on the path to  $s$  from the deepest node in  $\bar{T}$  that is a suffix of  $s$ ;

- (C) If  $|s| < L$  then for every  $\sigma' \in \Sigma$ , if

$$\tilde{P}(\sigma' \cdot s) \geq P_{min},$$

then add  $\sigma' \cdot s$  to  $\bar{S}$ .

3. Smoothing the prediction probabilities: For each  $s$  labeling a node in  $\bar{T}$ , let  $\bar{\gamma}_s(\sigma) = \bar{P}(\sigma|s)(1 - |\Sigma|\gamma_{min}) + \gamma_{min}$

Figure 1: Learning a Variable-Memory Probabilistic Automaton

Given a string, its weighting by a tree is done by scanning the string one symbol after the other while assigning a probability to every symbol, in succession. The probability of a symbol is calculated by walking down the tree in search for the longest suffix that appears in the tree and ends immediately before that symbol, and multiplying the corresponding conditional probability. Since, following each input symbol, the search for the deepest node must be resumed from the root, this process cannot be carried out on-line nor in linear-time in the length of the tested sequence.

Figure 1 reproduces for our convenience the construction of the tree from [7]. This is a re-scheduling of the algorithm of [15], which is equivalent for our purposes and thus will not be reproduced. Here  $L$  is the maximum length for a string to be considered,  $P_{min}$  is the minimum value of the empirical probability in order for the string to be considered,  $r > 1$  measures the prediction difference between the candidate and its father,  $\alpha$  the  $\epsilon$ 's and  $\gamma_{min}$  are suitable constants and part of the input parameters.

We are interested primarily in the asymptotic complexity of the main part (tree construction) of these procedures, and in possible ways to improve it. In this regard, the algorithms may be considered equivalent except for small changes in the treatment of internal nodes and the choice of constants. These details have no substantial bearing on the performance and no consequence on our considerations. Also, for the same reason, we are not interested in the last steps of smoothing probabilities. We see that the body of the algorithm of Figure 1 consists of checking all substrings having empirical probability at least  $P_{min}$  and length at most to  $L$ . Although the

number of substrings passing these tests may be quite small in practice, still there are in principle  $n - l + 1$  possible different strings for each  $l$ , which would lead to  $\Theta(Ln^2)$  time just to compute and test empirical probabilities ( $nL$  strings in total each requiring at least  $O(n)$  work to test). The discussion that follows shows that in fact overall  $O(n)$  time suffices. Along the way, an alternative notion of empirical probability is proposed and computed. We assume henceforth that  $S$  contains only one string, which will be denoted by  $x$ .

Our approach must depart considerably from the algorithm of the figure. There, word selection and tree construction go hand in hand in Step B. In our case, even though in the end the two can be re-combined, we decouple these tasks. We concentrate on word selection, hence on the tests of Step B. Essentially, we want to show that all those words can be tested in overall linear time, even if those word lengths may add up to more than linear space. Throughout, we reason to fix the ideas in terms only of the algorithm of Figure 1, since the other one may be handled analogously.

### 3 Computing Empirical Probabilities

Consider for a moment the problem of defining and computing empirical probabilities. This is needed at the beginning and intermediate steps of the algorithm, and also to dynamically update the set  $S$  while keeping its size from growing exponentially. One problem here is that the notion of empirical probability is not straightforward. Fortunately in the algorithm—in so far as the construction of the automaton goes—we are interested primarily in *conditional* probabilities which turn out to be less controversial.

One ingredient in the computation of empirical probabilities is the count of occurrences of a string in another string or set of strings. We concentrate on this problem first. Since there can be  $O(n^2)$  distinct substrings in a string of  $n$  symbols, a table storing the number of occurrences of all substrings of the string might take up in principle  $\Theta(n^2)$  space. However, it is well known that linear time and space suffice to build an index suitable to return, for any string  $w$ , its  $\chi_w$  count in  $x$ . Here we need to analyze this fact a little closer. We begin by recalling an important “left-context” property, which is conveniently adapted from [9].

Given two words  $x$  and  $y$ , the *start-set* of  $y$  in  $x$  is the set of *occurrences* of  $y$  in  $x$ , i.e.,  $pos_x(y) = \{i : y = x_i \dots x_j\}$  for some  $i$  and  $j$ ,  $1 \leq i \leq j \leq n$ . Two strings  $y$  and  $z$  are equivalent on  $x$  if  $pos_x(y) = pos_x(z)$ . The equivalence relation instituted in this way is denoted by  $\equiv_x$  and partitions the set of all strings over  $\Sigma$  into equivalence classes. In the example of the string *abaababaabaababaababa*, for instance,  $\{ab, aba\}$  forms one such class and so does  $\{abaa, abaab, abaaba\}$ . Recall that the *index* of an equivalence relation is the number of equivalence classes in it. The following property is adapted from [9].

**Fact 3.1** The index  $k$  of the equivalence relation  $\equiv_x$  obeys  $k < 2n$ .

**Proof:** For any two substrings  $y$  and  $w$  of  $x$ , if  $C(w) \cap C(y)$  is not empty then  $y$  is a prefix of  $w$  or vice versa (i.e.,  $C(y) \subseteq C(w)$  or vice versa). Then, the containment

relation on  $C$ -classes forms a tree with  $|x|$  leaves and such that each internal node has degree at least 2, whence at most  $2|x| - 1$  nodes in total.  $\square$

Fact 3.1 suggests that we might restrict computation of empirical probabilities to the  $O(n)$  equivalence classes of  $\equiv_x$ . One incarnation of the tree evoked by the above proof—in fact, an alternate proof of its own—is the *suffix tree*  $T_x$  associated with  $x$ . We assume familiarity of the reader with the structure and its clever  $O(n \log |\Sigma|)$  time and linear space constructions such as in [12, 16, 18]. The word ending precisely at vertex  $\alpha$  of  $T_x$  is denoted by  $w(\alpha)$ . The vertex  $\alpha$  is called the *proper locus* of  $w(\alpha)$ . The *locus* of  $w$  is the unique vertex of  $T_x$  such that  $w$  is a prefix of  $w(\alpha)$  and  $w(\text{FATHER}(\alpha))$  is a proper prefix of  $w$ . One key element in the above constructions is in the following easy fact.

**Fact 3.2** If  $w = av$ ,  $a \in \Sigma$ , has a proper locus in  $T_x$ , then so does  $v$ .

To exploit this fact, *suffix links* are maintained in the tree that lead from the locus of each string  $av$  to the locus of its suffix  $v$ . Here we are interested in Fact 3.2 only for future reference. Having built the tree, some simple additional manipulations make it possible to count and locate the distinct (possibly overlapping) instances of any pattern  $w$  in  $x$  in  $O(|w|)$  steps. For instance, listing all occurrences of  $w$  in  $x$  is done in time proportional to  $|w|$  plus the total number of such occurrences, by reaching the locus of  $w$  and then visiting the subtree of  $T_x$  rooted at that locus. Alternatively, a trivial bottom-up computation on  $T_x$  can weigh each node of  $T_x$  with the number of leaves in the subtree rooted at that node. This weighted version serves then as a statistical index for  $x$ , in the sense that, for any  $w$ , we can find the count  $\chi_w$  of  $w$  in  $x$  in  $O(|w|)$  time. Note that the counter associated with the locus of a string reports its correct count even when the string terminates in the middle of an arc. This is, indeed, nothing but a re-statement and a proof of Fact 3.1. From now on, we assume that a tree with weighted nodes has been produced and is available for our constructions.

We are now ready to consider more carefully the notion of empirical probability. One way to define the empirical probability of  $w$  in  $x$  is to take the ratio of the count of the number  $\chi_w$  to  $|x| - |w| + 1$ , where the latter is interpreted as the maximum number of possible starting positions for  $w$  in  $x$ . For  $w$  and  $v$  much shorter than  $x$  we have that the difference between  $|x| - |w| + 1$  and  $|x| - |wv| + 1$  is negligible, which means that the probabilities computed in this way and relative to words that end in the middle of an arc do not change, i.e., we only need to compute those associated with strings that end at a node of the compact  $T_x$ .

This notion of empirical probability assumes that every position of  $x$  compatible with  $w$  length-wise is an equally likely candidate. This is not the case in general, since the maximum number of possible occurrences of one string within another string depends crucially on the compatibility of self-overlaps. For example, the pattern *aba* could occur at most once every two positions in *any* text, *abaab* once every four, etc. Compatible self-overlaps for a string  $z$  depend on the structure of the periods of  $z$ . We review these notions briefly.

A string  $z$  has a *period*  $w$  if  $z$  is a prefix of  $w^k$  for some integer  $k$ . Alternatively, a string  $w$  is a period of a string  $z$  if  $z = w^l v$  and  $v$  is a possibly empty prefix of  $w$ .

```

procedure maxborder ( y )
begin
  bord[0] ← -1; r ← -1;
  for m = 1 to h do
    while r ≥ 0 and yr+1 ≠ ym do
      r ← bord[r];
    endwhile
    r = r + 1; bord[m] = r
  endfor
end

```

Figure 2: Computing the longest borders for all prefixes of  $y$

Often when this causes no confusion, we will use the word “period” also to refer to the length or *size*  $|w|$  of a period  $w$  of  $z$ . A string may have several periods. The shortest period (or period length) of a string  $z$  is called *the period* of  $z$ . Clearly, a string is always a period of itself. This period is called the trivial period.

A germane notion is that of a border. We say that a non-empty string  $w$  is a *border* of a string  $z$  if  $z$  starts and ends with an occurrence of  $w$ . That is,  $z = uw$  and  $z = vw$  for some possibly empty strings  $u$  and  $v$ . Clearly, a string is always a border of itself. This border is called the trivial border. It is not difficult to see that two consecutive occurrences of a word may overlap only if their distance equals one of the periods of  $w$ . Along this line of reasoning, we have

**Fact 3.3** The maximum possible number of occurrences of a string  $w$  into another string  $x$  is equal to  $(|x| - |w| + 1)/|u|$ , where  $u$  is the smallest period of  $w$ .

If we wanted to compute the empirical probabilities of, say, all prefixes of a string along the definition of Fact 3.3, we would need first to know the borders or periods of all those prefixes. In fact, we can manage to carry out the computations relative to the set of prefixes of a same string in *overall* linear time, thus in amortized constant time per prefix. This possibility rests on a simple adaptation of a classical implement of fast string searching, that computes the longest borders (and corresponding periods) of all prefixes of a string in overall linear time and space. We report one such construction in Figure 2, for the convenience of the reader, but refer for details and proofs of linearity to discussions of “failure functions” and related constructs such as found in, e.g., [10].

The construction of Fig. 2 may be applied, in particular, to each suffix  $suf_i$  of a string  $x$  while that suffix is being inserted as part of the direct tree construction. This would result in an annotated version of  $T_x$  in overall quadratic time and space in the worst case. Note that, unlike in the case of empirical probabilities previously considered, the period –and thus also the empirical probabilities of Fact 3.3– may change along an arc of  $T_x$ , so that we may need to compute explicitly all  $\Theta(n^2)$  of them. However, if we were interested in such probabilities only at the nodes of

the tree, then these could still be computed in overall linear time. The key to this latter fact is to run a suitably adapted version of `maxborder` walking on suffix links “backward”, i.e., traversing them in their reverse direction, beginning at the root of  $T_x$  and then going deeper and deeper into the tree. One way to visualize this process is as follows. Imagine first the “co-tree” of  $T_x$  formed by the reversed suffix links: we can visit such a structure depth first and simultaneously run a procedure much similar to `maxborder` to assign periods to all nodes of  $T_x$ . Correctness rests on the fact that for any word  $w$  the periods of  $w$  and  $w^r$  coincide. We shall see shortly that in situations of interest to us we can limit computation to the nodes of  $T_x$ .

**Lemma 3.4** The set of empirical probabilities of all (short) words of  $x$  that have a proper locus in  $T_x$  can be computed in linear time and space.

## 4 Conditional Probabilities and Ratios Thereof

Consider now conditional empirical probabilities, which were defined as the ratio between the observed occurrences of  $s\sigma$  to the occurrences of  $s*$ . The first thing to note is that the value of this ratio persists along each arc of the tree, i.e.,

$$\tilde{P}(\sigma|s) = \chi_s/\chi_{s\sigma} = 1$$

for any word  $s$  ending in the middle of an arc of  $T_x$  and followed there by a symbol  $\sigma$ . Therefore, we know that every such word passes the first test under  $(B)$ , while continuation of  $s$  by any other symbol would have zero probability and thus fail. These words  $s$  have then some sort of an obvious implicit vector and need not be tested or considered explicitly. On the other hand, whenever both  $s$  and  $suffix(s)$  end in the middle of an arc, the ratio

$$\frac{\tilde{P}(\sigma|s)}{\tilde{P}(\sigma|suffix(s))} = \frac{1}{1} = 1.$$

Since  $r > 1$ , then neither  $r$  nor  $1/r$  may be equal to 1, so that no such word passes either part of the second test under  $B$ . The fate of any such word with respect to inclusion in the tree depends thus on that of its shortest extension with a proper locus in it. The cases where both  $s$  and  $suffix(s)$  have a proper locus in  $T_x$  are easy to handle, since there is only  $O(n)$  of them and the corresponding tests take linear time overall. By Fact 3.2, it is not possible that  $s$  has a proper locus while  $suffix(s)$  does not. Therefore, we are left with those cases where a proper locus exists for  $suffix(s)$  but not for  $s$ . There are still only  $O(n)$  such cases of course, but the problem here is that in these cases we do not find a suffix link to traverse backward from the locus of  $suffix(s)$  to that of  $s$ . Fortunately, this is easy to fix, but towards this end we need first to define reverse suffix links more formally.

Let  $\nu'$  be the locus of string  $s'$ . We define  $rsuf(\nu', \rho)$  as the node  $\nu$  which is the locus of the shortest extension of  $\rho s'$  having a proper locus in  $T_x$ . I.e., node  $\nu$  is such that  $w(\nu) = sz$  with  $s = \rho s'$  and  $z$  as short as possible if  $z \neq \lambda$ . If  $\rho s'$  has no occurrence in  $x$  then  $rsuf(\nu', \rho)$  is not defined.

Clearly,  $\text{rsuf}$  coincides with the reverse of the suffix link whenever the latter is defined. When no such original suffix link is defined while  $\rho s'$  occurs in  $x$ , then  $\text{rsuf}$  takes us to the locus of the shortest word in the form  $\rho s'z$ . Since  $\nu'$  is a branching node, then there are occurrences of  $s'$  in  $x$  that are not followed by the first character of  $z$ . In other words, not all occurrences of  $s'$  occur precisely at the second position of an occurrence of  $sz = \rho s'z$ , whence  $\text{pos}_x(\rho s') \neq \text{pos}_x(\rho s'z)$ . In these cases, we know *a priori* that  $\tilde{P}(\sigma|s) = 1$  only for  $\sigma$  equal to the first character of  $z$ , but the value of  $\tilde{P}(\sigma|s')$  and hence also of the ratio  $\tilde{P}(\sigma|s)/\tilde{P}(\sigma|s')$  have to be computed and tested explicitly. In these cases, the locus of  $\rho s'z$  may be treated also as a surrogate locus of that of  $\rho s'$ . Clearly, the number of such surrogates and their additional weights is linear in  $n$ .

Setting  $\text{rsuf}$ 's is an easy linear post-processing of  $T_x$ . Details are left for an exercise, or we refer to the discussion in [10] of the relationship between suffix trees and automata. In conclusion, attaching empirical conditional probabilities only to the branching nodes of  $T_x$  suffices. As there are  $O(n)$  such nodes, and the alphabet is finite, the collection of all conditional probability vectors for *all* subwords of  $x$  takes only linear space.

**Lemma 4.1** The set of empirical conditional probabilities of all (short) words of a string  $x$  over a finite alphabet can be computed in linear time and space. The conditions under Step B can be tested implicitly for all such words in overall linear time and space.

## 5 PSAs and MPMs

At this point we can informally summarize as follows the procedure by which words are selected for inclusion in our automaton.

1. Build a compact  $T_x$  for  $x$ , weigh its ( $O(n)$ , branching) nodes with the  $\chi$ -counts and empirical probabilities;
2. Determine the words to be included in  $\bar{S}$  and thus in the final automaton. For this, consider the nodes of  $T_x$  bottom up and run the tests of Step B on these nodes. Mark the root and all nodes passing the test. For every node marked, follow the path of suffix links to the root and mark all currently unmarked nodes on this path.
3. Prune the tree below the deepest nodes failing the tests. By the above Facts, we know that pruning will have to occur in correspondence with nodes, and not along an arc.

Let  $\mathcal{H}$  denote the pruned version of  $T_x$  resulting from this treatment.

**Theorem 5.1** The probabilistic automaton  $\mathcal{H}$  contains  $\mathcal{T}$  and all the information stored in  $\mathcal{T}$ , and can be learned in linear time and space.

The structure  $\mathcal{H}$  represents a Trie-shaped automaton capable of performing much the same function as a PSA and yet having the structure of a Multiple Pattern Matching Machine (MPMM) [2, 10]. The import of this is that, on such a machine, the paths corresponding to the substring undergoing test are traversed from the root to the leaves. The structure of MPMMs, with failure-function links etc., can be adapted in such a way that ensures that while the input string is scanned symbol after symbol, we are always at the node of the MPMM that corresponds to the longest suffix of the input having a node in the MPMM. Algorithms developed along these lines are presented in [4], and they lead to variants of MPMMs, such that running a string through them for classification or encoding takes always overall linear time in the length of that string.

**Acknowledgements** I am indebted to Andreas Dress for providing an inspiring atmosphere and stimulating discussions near the slopes of Klosters, to Gill Bejerano and Golan Yona for a careful scrutiny of the draft paper and many helpful comments.

## References

- [1] Abe, N. and M. Warmuth. On the Computational Complexity of Approximating Distributions by Probabilistic Automata, *Machine Learning*, 9, 205–260 (1992).
- [2] Aho, A.V. and M.E. Corasick, Efficient String Matching: an Aid to Bibliographic Search, *CACM*, 18, 333–340 (1975).
- [3] Apostolico, A., and Z. Galil (Eds.), *Pattern Matching Algorithms*, Oxford University Press, New York (1997).
- [4] Apostolico, A. and G. Bejerano, Optimal Amnesic Probabilistic Automata or How to Learn and Classify Proteins in Linear Time and Space, manuscript (1999).
- [5] Apostolico, A., M. E. Bock, and S. Lonardi, Linear Global Detectors of Redundant and Rare Substrings. In J. A. Storer and M. Cohn, editors, *Proceedings of Data Compression Conference*, 168–177, Snowbird, Utah, (April 1999).
- [6] Apostolico, A., M. E. Bock, S. Lonardi and X. Xu. Efficient Detection of Unusual Words, Technical Report 97-050, Purdue University Computer Science Department (1996). *Journal of Computational Biology*, to appear.
- [7] Bejerano, G. and G. Yona, Modeling Protein Families Using Probabilistic Suffix Trees. *Proceedings of RECOMB99* (S. Istrail, P. Pevzner and M. Waterman, eds.), 15–24, Lyon, France, ACM Press (April 1999).
- [8] Bell, T.C., J.G. Cleary and I.H. Witten, *Text Compression*, Prentice Hall, Prentice Hall, N.J. (1990)

- [9] Blumer, A., J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen and J. Seiferas, The Smallest Automaton Recognizing the Subwords of a Text, *Theoretical Computer Science*, 40, 31-55 (1985).
- [10] Crochemore, M., and W. Rytter, *Text Algorithms*, Oxford University Press, New York (1994).
- [11] Forchhammet, S., and J. Rissanen, Coding with Partially Hidden Markov Models, *Proceedings of the IEEE Data Compression Conference, DCC95*, 92-101, Snowbird, Utah, 1995.
- [12] McCreight, E.M., A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262-272, April 1976.
- [13] Rissanen, J., A universal Data Compression System, *IEEE Trans. Inform. Theory* 29(5): 656-664 (1983).
- [14] Rissanen, J., Complexity of Strings in the Class of Markov Sources, *IEEE Trans. Inform. Theory* 32(4): 526-532 (1986).
- [15] Ron, D., Y. Singer and N. Tishby, The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length. *Machine Learning*, 25:117-150 (1996).
- [16] Ukkonen, E., On-line Construction of Suffix Trees. *Algorithmica*, 14(3):249-260, 1995.
- [17] Weinberger, M.J., A. Lempel and J. Ziv, A Sequential Algorithm for the Universal Coding of Finite Memory Sources, *IEEE Trans. Inform. Theory* 38: 1002-1014 (1982).
- [18] Weiner, P., Linear Pattern Matching Algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1-11, Washington, DC, 1973.
- [19] Willems, F.M.J., Y.M. Shtarkov and T.J. Tjalkens, The Context Tree Weighting Method: Basic Properties, *IEEE Trans. Inform. Theory* 29(5): 656-664 (1993).