JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 7, Numbers 3/4, 2000
Mary Ann Liebert, Inc.
Pp. 381–393

Optimal Amnesic Probabilistic Automata or How to Learn and Classify Proteins in Linear Time and Space

ALBERTO APOSTOLICO[1] and GILL BEJERANO[2]

ABSTRACT

Statistical modeling of sequences is a central paradigm of machine learning that finds multiple uses in computational molecular biology and many other domains. The probabilistic automata typically built in these contexts are subtended by uniform, fixed-memory Markov models. In practice, such automata tend to be unnecessarily bulky and computationally imposing both during their synthesis and use. Recently, D. Ron, Y. Singer, and N. Tishby built much more compact, tree-shaped variants of probabilistic automata under the assumption of an underlying Markov process of variable memory length. These variants, called Probabilistic Suffix Trees (*PST*s) were subsequently adapted by G. Bejerano and G. Yona and applied successfully to learning and prediction of protein families. 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. The main contribution of this paper is to introduce automata equivalent to PSTs but having the following properties:

- **Learning the automaton, for any L, takes $O(n)$ time.**
- **Prediction of a string of m symbols by the automaton takes $O(m)$ time.**

Along the way, the paper presents an evolving learning scheme and addresses notions of empirical probability and related efficient computation, which is a by-product possibly of more general interest.

Key words: amnesic automata, probabilistic suffix trees, variable memory Markovian models, protein families, protein classification.

[1]Department of Computer Sciences, Purdue University, West Lafayette, IN 47907 and Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy.
[2]School of Computer Science and Engineering, The Hebrew University, Jerusalem 91904, Israel.

1. INTRODUCTION

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 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 actually being 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., Abe and Warmuth [1992], Apostolico *et al.* [1999], Apostolico *et al.* [1996], Rissanen [1983], Ron *et al.* [1996] 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., Forchhammet and Rissanen [1995], Rissanen [1983, 1986] in the context of predictive and universal codes, and to Bejerano and Yona [1999] and Ron *et al.* [1996] 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 quickly 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 the next symbol given the last $L' > L$ symbols does not change appreciably. It is possible and customary to model these sources by Markov chains of order L, which denotes the maximum useful memory length. Even so, such automata tend in practice to be unnecessarily bulky and computationally imposing both during their synthesis and use. In Ron *et al.* (1996) much more compact, tree-shaped variants of probabilistic automata (called *Probabilistic Suffix Trees*, or *PST*s) are built which assume an underlying Markov process of variable memory length not exceeding some maximum L. The probability distribution 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 properties that, on one hand, learning the automaton takes $O(n)$ time, regardless of L, and on the other, prediction of a string of m symbols by the automaton takes $O(m)$ time. Along the way, we address notions of empirical probability and their efficient computation, possibly a by-product of more general interest.

We adopt definitions and notations from Bejerano and Yona (1999) and Ron *et al.* (1996) with which some familiarity is assumed. We deal with a (possibly singleton) collection S of strings over a finite alphabet Σ and use λ 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 $s = s_1 s_2 \ldots s_l$, 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 σ immediately after the string s is given by the ratio

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

where χ_w is the number of occurrences of string w in S and $s*$ is every single-symbol extension of s having an occurrence in S. Finally, *suffix*(s) denotes $s_2 s_3 \ldots s_l$.

We recall the structure of a PST, as described in Bejerano and Yona (1999) and Ron *et al.* (1996) (see Figure 1, recapped from Bejerano and Yona [1999]). 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 PST, $\overline{S}$ is the set of strings that we want to check or learn, and γ_s is the probability distribution over the next symbol associated with node s. The construction starts with a tree consisting of only the root node (i.e., the tree associated with λ) and adds paths as follows. For each substring s considered, it is

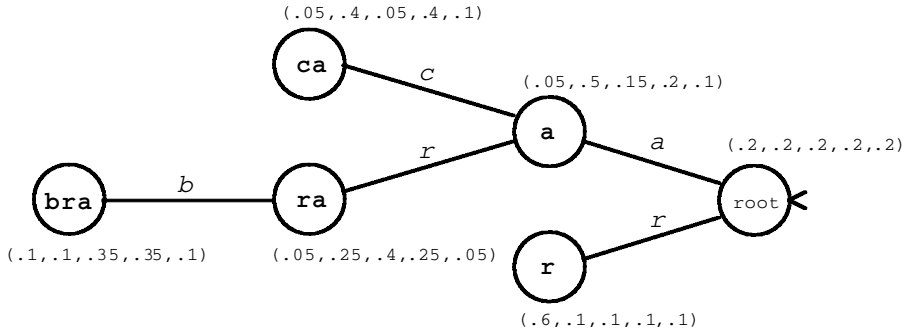

FIG. 1. An example of a PST over the alphabet $\Sigma = \{a, b, c, d, r\}$. The vector near each node is the probability distribution for the next symbol, e.g., the probability to observe c after a substring, whose largest suffix in the tree is ra, is 0.4. An exemplary prediction using this PST, denoted T:

$$
\begin{array}{rccccccc}
P^T(abracadabra) = & P^T(a) & P^T(b|a) & P^T(r|ab) & P^T(a|abr) & P^T(c|abra) & P^T(a|abrac) & \ldots \; P^T(a|abracadabr) \\
= & \overline{\gamma}_{root}(a) & \overline{\gamma}_a(b) & \overline{\gamma}_{root}(r) & \overline{\gamma}_r(a) & \overline{\gamma}_{bra}(c) & \overline{\gamma}_{root}(a) & \ldots \qquad \overline{\gamma}_r(a) \\
= & 0.2 & 0.5 & 0.2 & 0.6 & 0.35 & 0.2 & \ldots \qquad 0.6
\end{array}
$$

checked whether there is some symbol σ in the alphabet for which the empirical probability of observing it after s is both significant and significantly different from the probability of observing it after *suffix*(s). Whenever these conditions hold, the path relative to the substring (and possibly its necessary but currently missing ancestors) are added to the tree. As detailed below, the time complexity of this construction is $O(Ln^2)$, where L is the length of a longest string considered for possible inclusion in $\mathcal{T}$.

Given a string, its weighting or prediction by a PST is done by scanning the string one character after the other while assigning a probability to every character, in succession. The probability of a character is calculated by walking down the tree in search of the longest suffix that appears in the tree and ends immediately before that character; the corresponding conditional probability is then used in calculating the product for all characters (see Figure 1). 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 or in linear-time in the length of the tested sequence, the worst-case time being in fact $\Theta(m^2)$ for a sequence of m characters. In Ron *et al.* (1996, Appendix B) a solution is offered to this issue: a procedure is given to turn the PST into an equivalent and not-much-larger Probabilistic Finite Automaton (*PFA*) on which every prediction step does take constant time (is equal to a single transition on the PFA). However, this procedure may, by itself, cost $\Theta(Ln^2)$ time in the worst case.

In Bejerano and Yona (1999), and later, more extensively, in Bejerano and Yona (2000), the PST learning scheme is evaluated in the context of protein family modeling and classification, against an early version of the Pfam database (Bateman *et al.*, 2000). We briefly recount the results of this evaluation. The Pfam database uses expert knowledge and supervision along with multiple sequence alignments to train HMM models for families of related proteins. In Bejerano and Yona (2000), 170 such families, ranging from 884 to 10 members, were split randomly into a training and a test set in ratio 4 to 1. It was then shown, as a proof of principle, that in a fully automatic manner, without the use of multiple alignments, using a uniform set of parameters for all PST models, the PSTs were able to detect overall nearly 91% of all true positives. In several cases, they have even out performed the HMM models hand crafted for these families.[1] In Figure 2 we recount the results for a typical PST, taken from Bejerano and Yona (1999).

2. LEARNING AUTOMATA IN LINEAR TIME

The PST learning algorithm given below reproduces for our convenience the construction of the tree from Bejerano and Yona (1999).

[1]This is possible as an HMM is sometimes forced to accommodate sequences known to belong to its related family of proteins by the expert growing it, even though those proteins may not be well recognized by the model itself.

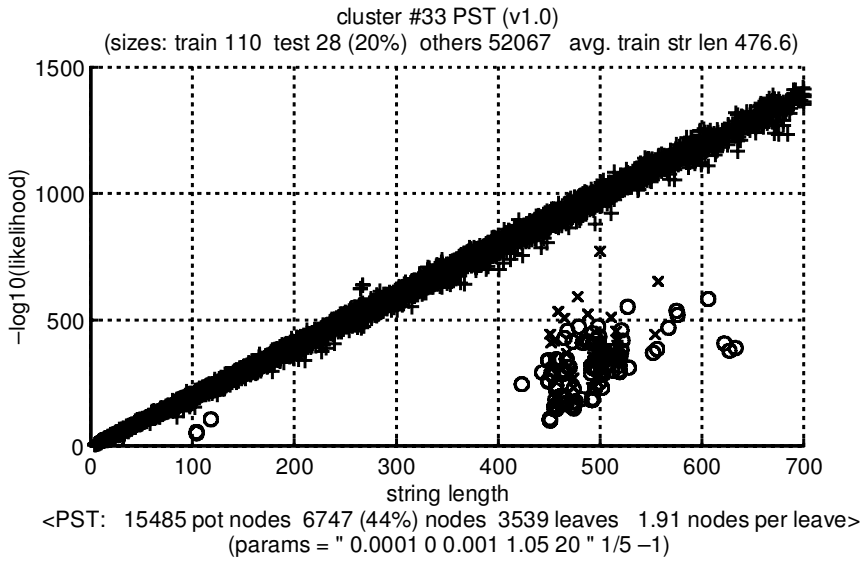

FIG. 2. Typical performance of a protein family PST, here of the neurotransmitter-gated ion-channels. Symbol "o" denotes training set samples, "x" denotes test set samples, and "+" denotes unrelated protein sequences, on a plot of protein sequence length vs. the score it has been assigned by a PST model trained only on the "o"s. One clearly sees that the model has generalized from the samples it has been shown, to "capture" the "x"s. The evaluation was done against the Swissprot database (Bairoch and Apweiler 2000), containing all protein sequences known at the time.

PST Learning Algorithm

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

$$\text{(Part I:)} \qquad \tilde{P}(\sigma|s) \geq (1+\alpha)\gamma_{min},$$

and

$$\text{(Part II:)} \qquad (1)\ \frac{\tilde{P}(\sigma|s)}{\tilde{P}(\sigma|suffix(s))} \geq r$$

or

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

   then add to $\overline{T}$ the node corresponding to s and all the nodes on the path to s from the deepest node in $\overline{T}$ 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 $\overline{S}$.
3. Smoothing the prediction probabilities:
   For each s labeling a node in $\overline{T}$, let $\overline{\gamma}_s(\sigma) = \tilde{P}(\sigma|s)(1 - |\Sigma|\gamma_{min}) + \gamma_{min}$.

This is a rescheduling of the algorithm of Ron *et al.* (1996), which is equivalent for our purposes and thus will not be reproduced. Here, L is, again, 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 multiplicative prediction difference between the candidate and its father for any given character, while α and γ_{min} limit the minimal empirical probability for a particular character to be of interest. The parameter γ_{min} is also used as the smoothing factor at the last stage of the construction.

We are interested primarily in the asymptotic complexity of the main part (tree construction) of the procedure and in possible ways to improve it. The last steps of smoothing probabilities have no substantial bearing on the performance and no consequence on our considerations. We see that the body of the algorithm consists of checking all substrings having empirical probability at least P_{min} and length at most L. Although the number of substrings passing these tests may be smaller in practice, 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 $\Theta(n)$work to test). The discussion that follows shows that, in fact, overall $O(n)$ time suffices.

Our approach must depart considerably from the algorithm of the figure. There, word selection and tree construction go hand in hand in Steps B and C. In our case, even though in the end the two can be recombined, 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. For simplicity of exposition we assume henceforth that S consists of only one string, which will be denoted by x.

2.1. Computing conditional probabilities and ratios thereof

A synopsis of tests on conditional probabilities is given in Figure 3. The goal of this subsection is to establish the following.

Lemma 2.1. *There is an algorithm to perform the collection of all tests under Step B for all substrings of S in overall linear time and space.*

Notice that S may contain $\Theta(n^2)$ distinct strings as substrings. Thus, there are two qualifications to the lemma: one is to show that computation can be limited to $O(n)$ words, the other is that this can be achieved in overall linear time and space. We now begin with the proof of the lemma.

Given two words x and y, the *start-set* of y in x is the set of *occurrences* of y in x, i.e., $pos_x(y) = \{i : y = x_i \ldots x_j\}$ 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, denoted by $\equiv_x$, partitions the set of all strings over Σ into equivalence classes. We use $C(w)$ to denote the equivalence class of w with respect to x. In the string $x = abaababaabaababaababa$, for instance, $\{ab, aba\}$ forms one such C-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 important "left-context" property is adapted from Blumer *et al.* (1985).

Fact 2.2. *The index k of the equivalence relation $\equiv_x$ obeys $k \leq 2n$.*

Proof. For any two substrings y and w of x, if $pos_x(w) \cap pos_x(y)$ is not empty then y is a prefix of w or vice versa (i.e., $(C(y) \subseteq C(w)$ or vice versa). If x is extended by appending to it a symbol not appearing anywhere else, then the containment relation on subsets of the form pos_x forms a tree with $|x| + 1$ leaves, each corresponding to a different position, and in which each internal node has degree at least 2. Therefore, there are at most $|x|$ internal nodes and $2|x| + 1$ nodes, or equivalence classes, in total. Taking back now the spurious leaf of position $(|x| + 1)$ yields the claim. ■

Locus of suffix(s)	*Locus of s*	*Action for Test I*	*Action for Test II*
(1) proper locus v'	proper locus v	read weights of locus v	use suffix link from v to v'
(2) middle of an arc	proper locus v	impossible	impossible (see Fact 2.3)
(3) proper locus v'	middle of an arc	single possible extension	use `rsuf` from v' to aux or surrogate locus of s
(4) middle of an arc	middle of an arc	irrelevant	always fails (see text)

FIG. 3. Synopsis of tests on conditional probabilities.

Fact 2.2 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 McCreight (1976), Ukkonen (1995), and Weiner (1973). The word ending precisely at vertex α of T_x is denoted by $w(\alpha)$. The vertex α is called the *proper locus* of $w(\alpha)$. The *locus* of word w is the unique vertex α 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 simple fact:

Fact 2.3. *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 2.3 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.

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 observation is that the value of this ratio persists along each arc of the T_x, 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 σ. 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 is

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

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 a PST (when also in the final version of our tree) would depend 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, as there is only $O(n)$ of them and the corresponding tests take linear time overall. By Fact 2.3, 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 in order to handle them we need first to perform a slight expansion on T_x.

Let v' be the proper locus of string s'. We define $\texttt{rsuf}(v', \rho)$ to be the node v, which is the proper locus of the shortest extension of $\rho s'$ having a proper locus in T_x. In other words, node v has the property that $w(v) = sz$ with $s = \rho s'$ and z as short as possible if $z \neq \lambda$ (see Figure 4a). If $\rho s'$ has no occurrence in x then $\texttt{rsuf}(v', \rho)$ is not defined.

Clearly, `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 `rsuf` takes us to the locus of the shortest word in the form $\rho s' z$. Since v' is a branching node, then there are occurrences of s' in x that are not followed

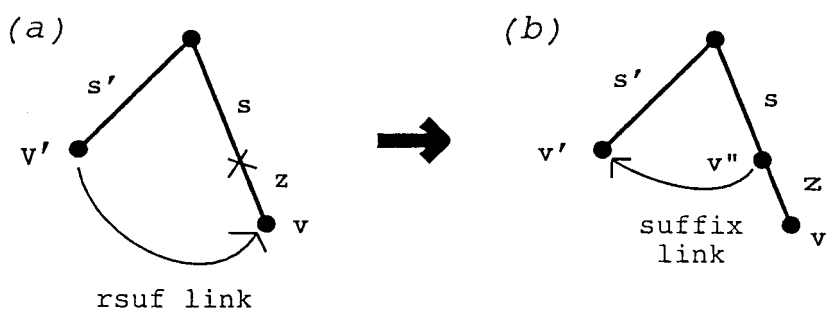

FIG. 4. Creating auxiliary suffix links. See text for details.

by the first character of z. In other words, not all occurrences of s' occur precisely at the second position of an occurrence of $sz = \rho s' z$, when $pos_x(\rho s') \neq pos_x(\rho s' z)$. In these cases, we know a priori that $\tilde{P}(\sigma|s) = 1$ only for σ 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. We can do so by treating v as a surrogate locus of that of $s = \rho s'$, but it is more convenient for our discussion to add to T_x explicit unary nodes for this purpose. Thus, a special unary node v'' is created as the proper locus of $\rho s'$ and endowed with a suffix link directed toward v' (see Fig. 4b). It should be clear that the total number of such auxiliary nodes in our tree is bounded by $n|\Sigma|$, and hence is linear for finite alphabets. Figure 3 summarizes the possible cases and their respective treatments.

Expanding T_x and computing `rsuf`'s is an easy linear postprocessing of the tree. We have also seen that 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. Given T_x, the computation of such probabilities is trivially done in linear time. With reference to Figure 3, the only tests to be taken are of type (1) and (3), and there are $O(n|\Sigma|)$ such tests of each kind, both associated with the nodes of the tree. Specifically, there are $|\Sigma|$ comparisons at the nodes v and v' under (1) and $|\Sigma|$ possible extensions of the words s' associated with nodes v' under (3).

This concludes the proof of Lemma 2.1. ■

2.2. *Building the new amnesic automaton*

At this point we can already outline an $O(n|\Sigma|)$ procedure by which words are selected for inclusion in our automaton and the automaton itself is built.

1. **Substrate preparation:** Build a compact suffix tree T_x for x. Add auxiliary unary nodes as described.
2. **Word selection:** Determine the words to be included in $\overline{S}$ and thus in the final automaton. For this, visit the nodes of T_x bottom up, compute χ-counts and conditional probabilities, 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 nodes on this path currently unmarked.
3. **Tree pruning:** Visit the tree in some bottom-up order, and prune the tree cutting all edges immediately below every deepest unmarked node.

In practice, the operations above would be more suitably arranged and combined without this affecting their global complexity. Let $\mathcal{H}$ denote the pruned version of T_x resulting from this treatment. As is easily seen, the set of words having proper loci at a marked node of $\mathcal{H}$ contains that of the words associated with the nodes of the PST resulting from the PST learning algorithm. In particular, the marking of all nodes on the suffix path of a marked node corresponds to admitting into the tree all suffixes of every admitted word.

Fact 2.4. *If word w is spelled out on some path from a node to the root of the PST $\mathcal{T}$, then w has a marked proper locus in $\mathcal{H}$.*

This fact shows just how the PST $\mathcal{T}$ is embedded in $\mathcal{H}$: to extract $\mathcal{T}$ from $\mathcal{H}$, take the marked nodes of $\mathcal{H}$ and the `rsuf`s edges connecting these nodes and then possibly prune some fringes at the bottom of the tree thus obtained. Any marked node v in $\mathcal{H}$ that does not appear in $\mathcal{T}$ corresponds to a node that the PST algorithm would have inserted had it gotten to it (or to a marked descendant of it). However, due to the top-down nature of the PST algorithm combined with a possibly *nonmonotone* notion of $\tilde{P}$, if any node along the `rsuf` path of v fails test C (even though v itself passes it), node v would never be examined by the PST algorithm.[2] One might argue that these nodes should have also been included in $\mathcal{T}$ and hence

[2]Note, for example, that the notion of $\tilde{P}$ we use is nonmonotone, i.e., there may be $w \in \Sigma^\star$ and $\sigma \in \Sigma$ s.t. $\tilde{P}(w) < \tilde{P}(\sigma w)$. Consider the case where $\Sigma = \{a, b\}$ and $x = baabaa$. A simple calculation shows that $0.4 = \tilde{P}(aa) < \tilde{P}(baa) = 0.5$. This means that if the threshold in Test C is set to 0.45 the node corresponding to baa will not be examined for inclusion in $\mathcal{T}$, because its father node, corresponding to aa, will fail to pass test C. However, as $\mathcal{H}$ prunes bottom-up, it will encounter the node corresponding to baa. This node may very well pass test B, and as a result both it *and* its father will be included in $\mathcal{H}$.

must stay, or modify the pruning of T_x so that these nodes are excluded from $\mathcal{H}$ as well (this requires one walk on `rsuf`s). Yet another alternative is to defer the tests of Step C to the weighting phase, in which they may be performed on the fly, where desired, without this affecting the time complexity of that phase. This issue shall be further discussed in Section 5.

Essentially, $\mathcal{H}$ is a compact trie resembling the basic structure of a multiple pattern matching machine (MPMM) (Aho and Corasick, 1975). The import of this is that, on such a machine, substrings undergoing tests are scanned in the forward, rather than reverse, direction while traveling on paths that go from the root to the leaves of the automaton. The full-fledged structure of MPMMs, with failure-function links, etc., 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. Also, by the structure of MPMMs, running a string through it always takes overall linear time in the length of the string. However, our MPMM is nonstandard in that explicit nodes (and associated failure pointers) might be missing along the arcs of T_x. The total number of such nodes might amount to $\Theta(n^2)$ in the worst case. One might consider adding such nodes on the fly during prediction at a cost of constant time per character and charge the predicted sequence(s) with the corresponding $O(m)$ work. In the next section, we study means of surrogating the missing nodes and links within the $O(n)$ time allocated to learning. We conclude this section by recording the following

Theorem 2.5. *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.*

3. IMPLEMENTING LINEAR TIME PREDICTORS

In this section, we assume we are given a pruned tree $\mathcal{H}$ with its nodes suitably weighted and marked, and we tackle the problem of how to use this tree for prediction. We consider two different scenarios for prediction, depending on whether the string s is assumed to be fed to $\mathcal{H}$ one character at a time from left to right or backward, beginning with the last character. We first sketch our treatment of the first case and then discuss the second one in full detail.

In a left-to-right scanning, we want to maintain that at the generic step where we have read the prefix $s_1 s_2 \ldots s_{j-1}$ we find ourselves at the marked node v of $\mathcal{H}$ that is the proper locus for the longest suffix of $s_1 s_2 \ldots s_{j-1}$ among those suffixes that have a marked proper locus in $\mathcal{H}$.

Let us say that a node μ has a *direct* σ-child in $\mathcal{H}$ if μ has a child node μ' reachable through an edge labeled only by the character $\sigma \in \Sigma$. Node μ is then the *direct* father of μ'. Back to the discussion, our approach distinguishes two cases, depending on whether or not the node v has a direct s_j-child. We consider first the case where v does not have a direct s_j-child. This is the easier case, as the following lemma gives the handle for it.

Lemma 3.1. *Let $w(v) = s_f s_{f+1} \ldots s_{j-1}$ and $w(\mu) = s_{f'} s_{f'+1} \ldots s_j$ be the longest suffixes of $s_1 s_2 \ldots s_{j-1}$ and $s_1 s_2 \ldots s_j$, respectively, having a marked proper locus in $\mathcal{H}$. If v has no marked direct s_j-child, then $f' > f$.*

Proof. The condition $f = f'$ is impossible, as the only way for this to happen would be if v had a marked direct s_j-child. Since this is denied by hypothesis, then we are left with one of the following three possibilities: there is an edge to a child v' of v labeled by a string that begins with s_j but consists of more than one character; there is no edge from v whose label begins by s_j altogether; v had a direct s_j-child v' but v' is not marked. For each of these cases, we have to look elsewhere in $\mathcal{H}$ than among the children of v to find the deepest possible marked proper locus μ of a suffix of $s_1 s_2 \ldots s_j$. Assume now for a contradiction that we found μ such that $f > f'$. Since μ must be a marked node in $\mathcal{H}$ then so must be by construction all nodes that are proper loci of the suffixes of $w(\mu) = s_{f'} s_{f'+1} \ldots s_j$. Among these nodes, we find, in particular, the marked proper locus of $s_f s_{f+1} \ldots s_j$. But then v has a direct s_j-child, which contradicts the hypothesis (see Fig. 5a). ■

Consider now our second case, in which v has a marked direct s_j-child v' in $\mathcal{H}$. This case is trivial to handle whenever v' cannot be reached from a marked node through a path of suffix links labeled by some suffix of $s_1 \ldots s_{f-1}$. Indeed, if no such path exists then traversing the edge to v' propagates our invariant

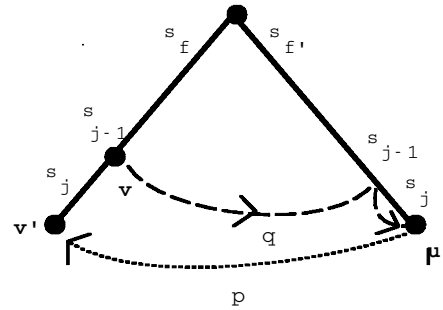

FIG. 5. Adding direct auxiliary links. (a) If v' does not exist and yet $f > f'$ then the suffix path p leads us to a contradiction. (b) Thus, if v' does not exist, it must be that $f < f'$. We wish then to hold a link from v to μ, the end point of path q. (c) Finally, if v' exists, we wish to hold a link from v' to μ, the end point of the reverse suffix path p^{-1}. Refer to the text for details.

condition to $s_1 s_2 \ldots s_j$, in constant time. If, on the other hand, such a path does exist, then the node on the longest possible such path is the node μ that we are seeking. Note that node μ depends on the structure of s and does not necessarily coincide with the deepest marked node encountered on an `rsuf` path from v'.

We now outline the computations involved in prediction when s is fed to $\mathcal{H}$ one symbol at a time from left to right. The cases contemplated in Lemma 3.1 are handled in constant time per symbol if we add to $\mathcal{H}$ links from every marked node and alphabet symbol to the closest node reachable by a number of direct transitions on suffix links followed by exactly one transition on a direct downward tree edge (see Fig. 5b). These links are easy to set in time linear in the size of $\mathcal{H}$.

To handle the case of a marked direct child node v' of v, we need to access, on the fly, the deepest marked node μ that is found on a reverse suffix path from node v' above and such that $w(\mu)$ corresponds to a suffix of $s_1 s_2 \ldots s_j$ (see Fig. 5c). This is made possible by a preprocessing on s which consists of running a multiple pattern matching for the (longest) substrings ending at marked nodes of $\mathcal{H}$. For this, $\mathcal{H}$ itself is suitably adapted (in linear time) in order to be treated as a standard MPMM. The information collected in this way is used during the weighting stage. Intuitively, we use the tracks left behind by s on its trail in the MPMM, and this enables us now to locate, in constant time, the deepest $rsuf$ descendant of v' which is compatible with a suffix of $s_1 s_2 \ldots s_j$. The net worth is that now there is one transition to the appropriate marked node for every symbol of s, when s is weighted in linear time.

Note that $\mathcal{H}$ is in compact form so that specifying failure transitions on it while keeping the $O(n)$ time and space is not obvious. The details are deferred to a forthcoming paper. In what follows, we concentrate on the alternative assumption that s is available *off-line* so that it can be fed *backwards* to $\mathcal{H}$. Since we are interested only in the product of all subterms, the order in which they are calculated may be altered at will. We show a simple and elegant linear time prediction phase that works for this case.

Theorem 3.2. *Given the automaton $\mathcal{H}$, there is an algorithm to weight any string s in overall $O(|s|)$ time.*

Proof. We retain the preprocessing that assigns to every node v a pointer to the closest marked node μ that can be reached following suffix links from v (Fig. 5b). The bulk of the algorithm consists of walking on the `rsuf` links of $\mathcal{H}$ in response to consecutive symbols of $s^R = s_m s_{m-1} \ldots s_1$, making occasional steps "sideways" along an edge of $\mathcal{H}$. The work is partitioned in *batches* of operations where each batch advances our knowledge of the deepest marked nodes for a certain number of suffixes of s^R. Each batch is associated with a substring of s^R and the work it performs is linear in that substring. Consecutive batches parse s^R into consecutive nonoverlapping substrings of s^R, when the linear overall bound. Batches are issued at a subset of the set of positions of s^R, and each batch faces a primary task and a maintenance task. If a batch is invoked in connection with $s_j s_{j-1} \ldots s_1$, the primary task of the batch is to find the two nodes $\mu = reach(j)$ and $v = mark(j)$ which correspond, respectively, to the deepest and deepest-marked node on the path of `rsuf`s from the root that is labeled by a prefix of $s_j s_{j-1} \ldots s_1$. A by-product of the primary task is to weight symbol s_j. The maintenance task is explained in what follows.

The batch for j starts having being handed a node $\theta = start(j)$ on the `rsuf` path for $s_j s_{j-1} \ldots s_1$ (consult Figure 6a). Let $s_j s_{j-1} \ldots s_h$ be the word labeling the `rsuf` path from the root to node θ and

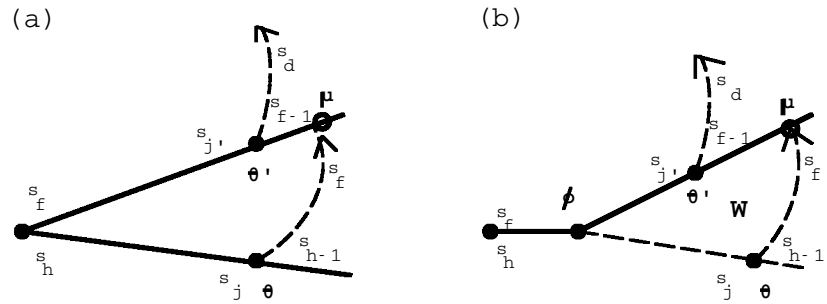

FIG. 6. Parsing a batch in backwards mode. The text holds the full details.

consider the path P of original T_x edges that connect the root of $\mathcal{H}$ to θ. Prior to inception of this batch, the following conditions hold.

1. For all suffixes $s_k s_{k-1} \ldots s_1$ with $k > j$, $mark(k)$ is already known.
2. Consider the collection of all $rsuf$ paths that are defined by walking from the root of $\mathcal{H}$ until the path ends or a node of P is met (each such path being the path or a prefix of the path to $mark(k)$ for $j \geq k \geq h$). For $k = j, j-1, \ldots h$, $mark(k)$ is currently set to the deepest marked node on its corresponding path.

The work begins at θ by following `rsuf`s while parsing the symbols that follow s_h in s^R until node μ is found. The algorithm climbs to $\theta' = \text{FATHER}(\mu)$, the father node of μ, which will be passed on to the next batch where it will take the place of θ. The string $\overline{s} = s_{h-1} s_{h-2} \ldots s_f$, connecting θ to μ, is the substring of s^R contributed by this batch to the parse of s^R mentioned above. At this point, we know the final value of $mark(j)$, as this must be either the deepest node known when the batch began or the deepest node encountered while scanning $\overline{s}$.

As $w(\text{FATHER}(\mu))$ is a prefix of $w(\mu)$, then the `rsuf` path from the root to $\text{FATHER}(\mu)$ corresponds to some suffix $s_{j'} s_{j'-1} \ldots s_f$ of $s_j s_{j-1} \ldots s_f$. The scan beginning at $start(j') = \text{FATHER}(\mu)$ will map into a substring $s_{f-1} s_{f-2} \ldots s_d$ of s^R that immediately follows $\overline{s}$ and thus has no overlap with this string. Before the new batch at j' can begin, however, Invariants 1 and 2 must be restored. We thus address the maintenance task of the batch.

Let ϕ be the lowest common ancestor of μ and θ in $\mathcal{H}$ (consult Figure 6b). The only nodes where the invariants might have been infringed are those found in the subtree W of $\mathcal{H}$, which is rooted at ϕ and has leaves at the nodes encountered on the path of $\overline{s}$ from θ to μ. The invariants are restored by visiting this subtree and checking, for every node in it, whether the pointer to the closest marked node gives an improvement over the current corresponding value of *mark*. Application of an argument already used in Lemma 3.1 to nodes μ and $\text{FATHER}(\mu)$ shows that, in particular, this treatment propagates Invariant 1 to all values of k in the interval $[j, j')$, i.e., Invariant 1 now holds for all $k > j'$. The number of nodes encountered in the visit is bounded by $|\overline{s}|$, the number of leaves, when the work involved is linear in $|\overline{s}|$. ■

4. COMPUTING EMPIRICAL PROBABILITIES

We consider here, in greater detail, the notion of empirical probability for a string and its related computations. This notion is not straightforward. Fortunately, in the algorithm—insofar 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. As was shown, although there can be $\Theta(n^2)$ distinct substrings in a string of n symbols, Fact 2.2 and the very structure of T_x show that linear time and space suffice to build an implicit table of χ_w counts of all strings w in x.

One way to define the empirical probability $\tilde{P}(w)$ of w in x is to take the ratio of the count χ_w to $|x| - |w| + 1$, where the latter is interpreted as the maximum number of possible starting positions for w

in x. This corresponds to viewing $\tilde{P}(w)$ as χ_w divided by $\chi_{|w|}$ (i.e., how many of the overall $n - l + 1$ substrings of length l were actually w).[3] From the computational standpoint, for w and v much shorter than x, we have that the difference between $|x| - |w| + 1$ and $|x| - |wv| + 1$ is negligible (this is not automatically true for any set S of k strings, where we would have $|x| - k|wv| + 1$), which means that the probabilities computed in this way and relative to words that end in the middle of an arc of T_x do not change, i.e., computing probabilities for strings with a proper locus is enough to know the probabilities of all substrings.

This notion of empirical probability, however, assumes that every position of x compatible with w lengthwise is an equally likely candidate. This is not the case in general, as the maximum number of possible occurrences of one string within another string crucially depends 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. 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. When this causes no confusion, we will use the word "period" to refer also 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. A string is trivially always a period of itself. 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 4.1. *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 4.1, we would first need to know the periods of all those prefixes. In fact, by a classical result of string matching, the period computations relative to the set of prefixes of a same string can be carried out in *overall* linear time, thus in amortized constant time per prefix. We refer for details and proofs to, e.g., Aho and Corasick (1975) and Apostolico and Galil (1997). Such a construction 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.

Perhaps more interestingly, we have that for empirical probabilities defined by Fact 4.1 the following holds.

Theorem 4.2. *The set of values $\tilde{P}(w) = \chi_w \cdot |u|/(|x| - |w| + 1)$ can be computed for all words of x that have a proper locus in T_x in overall linear time and space.*

Proof. Simply compute periods while 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. This walk intercepts all nodes of T_x. Correctness rests on the fact that for any word w the periods of w and w^r coincide. ■

Note, however, that since the period may vary in the middle of an arc, so could the empirical probabilities computed in this way. This weakens the assumption that the probability of a short word ending in the middle of an arc is surrogated by that of the shortest extension of that word with a proper locus. Fortunately, the discussion that led to Fact 2.4 shows that $\mathcal{T}$, being nothing but a subgraph of $\mathcal{H}$ connected by `rsuf`s, only needs the $O(n)$ probabilities at the $O(n)$ nodes of $\mathcal{H}$, irrespective of how such probabilities are defined.

5. FINAL REMARKS

Using the known duality between direct and reverse suffix links, it is natural to revolve our previous construction around and learn trees for the reverse of the strings in set S. Indeed, the PST tree structure

[3] This definition has a convenient probabilistic quality in that $\forall l = 1, 2, \ldots, L \sum_{|w|=l} \tilde{P}(w) = 1$.

itself is but a subtree of the expanded suffix tree of S^R. In such a dual construction, the learning phase is concerned with building a suitable, reverse-annotated tree of x^R while the weighting phase will traverse this tree.

Another important aspect to be analyzed in a forthcoming paper concerns setting up procedures of unsupervised learning that can follow some initial training phase. Once some version of the automaton is constructed from an initial set of positive examples, one wishes to easily learn a new example, i.e., update it in linear time, in the very same manner all previous examples in S were assimilated one by one. The same goes for removing a sequence from our pool. Along these lines, we develop a simple incremental learning scheme. Start off with some initial seed S from which the concept is first built. Then, while predicting over query sequences, one may, when coming across a sequence that, with high likelihood, belongs to the family, efficiently assimilate it into the learned structure before proceeding. Similarly, one may from time to time go over the list of sequences composing S and check whether, due to the evolution of the concept, some members no longer fit the concept. These may then be efficiently rejected. This is a useful feature to have when learning from noised or error-prone data, as is our case.

Other closely related benefits stem from deliberately abstaining from pruning our trees or presmoothing the head count vectors implicit in them. These facts allow us to couple the incremental nature presented above with a *simulated annealing* schedule (see Kirkpatrick and Gelatt [1983]). Namely, we may start off with a rather permissive notion of a significant pattern and an appropriate smoothing technique, and gradually during learning, while we evolve our notion of a family and hopefully put it on firmer grounds, we may "cool down" our system by increasing the threshold for significance, while lowering the impact of smoothing. We may also alter L—now taken as the maximal prediction (but not learning) depth—similarly.

The main theme of this paper has been the optimization of the PST learning algorithm time and space complexity. This method, recently introduced in the context of Protein family modeling in Bejerano and Yona (1999), has already shown a potential in becoming a useful tool in tackling this hard problem, as well as the closely related problem of finding remote protein homologies. More extensive experimentation, presented in Bejerano and Yona (2000), further strengthens this notion. However, one of the main hinges along the way of a computational tool to become practicable by the bioinformatics community is its run time requirements. Prior to the work presented here, the learning algorithm, implemented in a quadratic manner, required some 1–2 hours of c.p.u. time on a strong Pentium machine. Preliminary experiments indicate that much faster algorithms result from implementation of our work. Finally, we believe that the related notions presented here of empirical significance measures and of concept evolution will open the way to more fruitful investigations.

ACKNOWLEDGMENTS

The authors would like to thank Andreas Dress and Golan Yona for many useful comments. A.A. would also like to thank Manfred Eigen and all participants in his 34th Winterseminar for providing a congenial atmosphere and for stimulating discussions near the slopes of Klosters. G.B. would also like to thank Nir Friedman for getting him started on efficiency matters and Naftali Tishby for enlightening discussions.
A.A. was supported in part by NSF Grant CCR-9700276 and by the Italian Ministry of Research. G.B. was supported by a grant from the Ministry of Science, Israel.

REFERENCES

Abe, N., and Warmuth, M. 1992. On the computational complexity of approximating distributions by probabilistic automata, *Machine Learning* 9, 205–260.

Aho, A.V., and Corasick, M.E. 1975. Efficient string matching: An aid to bibliographic search, *CACM* 18, 333–340.

Apostolico, A., 2000. Notes on learning probabilistic automata, technical report 99-028, Purdue University Computer Science Department (September '99), abstracted in *Proceedings of DCC 2000*, 545–545, Snowbird, IEEE Press.

Apostolico, A., and Bejerano, G. 2000. Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space, *Proceedings of RECOMB 2000*, 25–32, ACM Press, Tokyo.

Apostolico, A., Bock, M.E., and Lonardi, S. 1999. Linear global detectors of redundant and rare substrings. *Proceedings of DCC 1999*, 168–177, Snowbird, IEEE Press.

Apostolico, A., Bock, M.E., Lonardi, S., and Xu, X. 2000. Efficient detection of unusual words. *J. Comp. Biol.* 7(1/2), 71–94.

Apostolico, A., and Galil, Z. (eds.) 1997. *Pattern Matching Algorithms*, Oxford University Press, New York.

Bairoch A., and Apweiler, R. (2000). The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucleic Acids Research* 28(1), 45–48.

Bateman, A., Birney, E., Durbin, R., Eddy, S.R., Howe, K.L., and Sonnhammer, E.L. 2000. The Pfam Protein Families Database, *Nucleic Acids Research* 28, 263–266.

Bejerano, G., and Yona, G. 1999. Modeling protein families using probabilistic suffix trees. *Proceedings of RECOMB 99*, 15–24, Lyon, ACM Press.

Bejerano, G., and Yona, G., to appear. Variations on probabilistic suffix trees—A new tool for statistical modeling and prediction of protein families. *Bioinformatics*.

Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T., and Seiferas, J. 1985. The smallest automaton recognizing the sub-words of a text, *Theoretical Computer Science* 40, 31–55.

Forchhammet, S., and Rissanen, J. 1995. Coding with partially hidden Markov models, *Proceedings of DCC 1995*, 92–101, Snowbird, IEEE Press.

Kirkpatrick, S., and Gelatt, C.D., Jr., 1983. Optimization by simulated annealing. *Science* 220, 671–680.

McCreight, E.M., 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM* 23(2), 262–272.

Rissanen, J., 1983. A universal data compression system, *IEEE Trans. Inform. Theory* 29(5), 656–664.

Rissanen, J., 1986. Complexity of strings in the class of Markov sources, *IEEE Trans. Inform. Theory* 32(4), 526–532.

Ron, D., Singer, Y., and Tishby, N. 1996. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning* 25, 117–149.

Ukkonen, E., 1995. On-line construction of suffix trees. *Algorithmica* 14(3), 249–260.

Weiner, P., 1973. Linear Pattern Matching Algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, 1–11, IEEE Press, Washington DC.

Address correspondence to:
Alberto Apostolico
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

E-mail: axa@cs.purdue.edu