BIOINFORMATICS ORIGINAL PAPER

Vol. 28 no. 10 2012, pages 1314–1323
doi:10.1093/bioinformatics/bts121

Sequence analysis

Advance Access publication April 19, 2012

Probabilistic suffix array: efficient modeling and prediction of protein families

Jie Lin[1], Donald Adjeroh[2,*] and Bing-Hua Jiang[3]

[1]Faculty of Software, Fujian Normal University, Fuzhou 350108, China, [2]Lane Department of Computer Science & Elect. Engineering, West Virginia University, Morgantown, WV 26506 and [3]Department of Pathology, Anatomy & Cell Biology, Thomas Jefferson University, Philadelphia, PA 19107, USA

Associate Editor: Martin Bishop

ABSTRACT

Motivation: Markov models are very popular for analyzing complex sequences such as protein sequences, whose sources are unknown, or whose underlying statistical characteristics are not well understood. A major problem is the computational complexity involved with using Markov models, especially the exponential growth of their size with the order of the model. The probabilistic suffix tree (PST) and its improved variant sparse probabilistic suffix tree (SPST) have been proposed to address some of the key problems with Markov models. The use of the suffix tree, however, implies that the space requirement for the PST/SPST could still be high.

Results: We present the probabilistic suffix array (PSA), a data structure for representing information in variable length Markov chains. The PSA essentially encodes information in a Markov model by providing a time and space-efficient alternative to the PST/SPST. Given a sequence of length N, construction and learning in the PSA is done in $O(N)$ time and space, independent of the Markov order. Prediction using the PSA is performed in $O(m\log\frac{N}{|\Sigma|})$ time, where m is the pattern length, and Σ is the symbol alphabet. In terms of modeling and prediction accuracy, using protein families from Pfam 25.0, SPST and PSA produced similar results (SPST 89.82%, PSA 89.56%), but slightly lower than HMMER3 (92.55%). A modified algorithm for PSA prediction improved the performance to 91.7%, or just 0.79% from HMMER3 results. The average (maximum) practical construction space for the protein families tested was $21.58\pm6.32N$ (41.11N) bytes using the PSA, $27.55\pm13.16N$ (63.01N) bytes using SPST and $47\pm24.95N$ (140.3N) bytes for HMMER3. The PSA was 255 times faster to construct than the SPST, and 11 times faster than HMMER3.

Availability: http://www.csee.wvu.edu/~adjeroh/projects/PSA

Contact: don@csee.wvu.edu

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on July 3, 2011; revised on February 10, 2012; accepted on March 1, 2012

1 INTRODUCTION

Markov models are often used for modeling complex sequences, such as protein sequences, whose underlying statistical characteristics are not well understood. This is especially the case when the sequences exhibit short-term memory. For a short-term memory of length, say L, the sequences can be modeled using Markov models of order L, or using Hidden Markov Models (HMMs; Ephraim *et al.* (1989). The models provide efficient mechanisms to compute the required conditional probabilities, and also for generating sequences from the models. The problem is that the size of Markov models increases exponentially with increasing memory length L. Thus, they are practical only for low-order models with short-memory lengths. This leads to another problem: such low-order Markov models often provide a poor approximation of the true sequence being modeled. It is known that learning and inference with HMMs is computationally very challenging (Abe and Warmuth, 1992; Gillman and Sipser, 1994).

Probabilistic suffix models such as probabilistic suffix trees (PSTs) have been proposed by Ron *et al.* (1996) to address some of the key problems with Markov models. They showed the equivalence between PSTs and a subclass of probabilistic finite automata (PFA) called probabilistic suffix automata (PFA/PSA): for a given PST, there is an algorithm to construct a PFA/PSA whose size is the same as that of the PST within a constant factor. Further, the distribution generated by a PST is guaranteed to be within a small distance from that generated using the PSF/PSA, as measured by the Kullback–Leibler divergence. Their probabilistic suffix models, however, require $O(LN^2)$ time and space to construct, where N is the sequence length. The algorithm to construct the PFA/PSA from the PST also runs in $O(LN^2)$ time. Later, (Apostolico and Bejerano, 2000) showed how the PST can be constructed in $O(N)$ time, independent of the order L, using traditional suffix links used in constructing suffix trees (STs), and the notion of reverse suffix links.

PSTs and probabilistic suffix automatons are related to context-based models which are extensively used in sequence prediction and data compression. The use of these context models as a surrogate for variable length Markov models with applications in sequence prediction are reviewed in (Begleiter *et al.*, 2004). Earlier, the PST was used to model DNA sequences in (Ron *et al.*, 1996). Modeling and prediction of protein families using the PST was reported in (Bejerano and Yona, 2001; Leonardi, 2006). Importantly, using the PST, prediction was performed without any prior alignment of the protein sequences. (Leonardi, 2006) proposed an improved variation of the PST, called sparse probabilistic suffix tree (SPST) for use in protein family prediction. Unlike the usual PST, the SPST allows some contexts to be grouped together to form an equivalent class. Essentially, this grouping results in a form of pruning of the PST, however, the node depths are not fixed, but could vary depending

*To whom correspondence should be addressed.

on the sequence. In Leonardi (2006), it was shown that the SPST resulted in an improved prediction accuracy, when compared with the PST.

The use of the ST, however, also implies that the practical time and space needed to construct or use the PST (or SPST) could be very high, although it is still theoretically linear in terms of the sequence length. Here, we present the PSA, a time and space-efficient alternative to the PST/SPST. The PSA provides all the capabilities of the PST/SPST, such as learning and prediction, and maintains the same linear time for construction. The PSA is, however, significantly more efficient than the SPST (in both space and time), for both the construction stage and the prediction stage.

2 BACKGROUND

2.1 Suffix trees and suffix arrays

Given a string $T=T[1,...,N]$ with symbol alphabet Σ, its ST is a rooted tree with N leaves, where the i-th leaf node corresponds to the i-th suffix T_i of T. Except for the root node and the leaf nodes, every node must have at least two descendant child nodes. Each edge in the ST represents a substring of T, and no two edges out of a node start with the same character. For a given edge, the *edge label* is simply the substring in T corresponding to the edge. We use l_i to denote the i-th leaf node. Then, l_i corresponds to T_i, the i-th suffix of T. When the edges from each node are sorted lexicographically, then l_i will correspond to $T_{\mathrm{SA}[i]}$, the i-th suffix of T in lexicographic order.

Some ST construction algorithms make use of *suffix links*. The notion of suffix links is based on a well-known fact about suffix trees (Gusfield, 1997; McCreight, 1976), namely, if there is an internal node u in ST such that its path label from the root $LL(u)=a\alpha$ for some single character $a\in\Sigma$, and a (possibly empty) string $\alpha\in\Sigma^*$, then there is a node v in ST such that $LL(v)=\alpha$. A pointer from node u to node v is called a *suffix link*. If α is an empty string, then the pointer goes from u to the root node. Suffix links are important in certain applications, such as in computing matching statistics needed in approximate pattern matching, regular expression matching or in certain types of traversal of the ST.

A predominant factor in the space cost for STs is the number of interior nodes in the tree, which depends on the tree topology. A simple implementation of the ST, for instance using Ukkonen's algorithm (Ukkonen, 1995), can require as large as 33N bytes of storage with suffix links, or 25N bytes without suffix links (Adjeroh *et al.*, 2008). Improved ST construction algorithms Kurtz (1999) require 15.67N bytes on average, and $21N-29N$ bytes in the worst case (includes 1 byte/symbol for input text), depending on whether linked lists or hash tables were used for representing the nodes. These do not include the extra space required for the auxiliary structures needed for the PST.

The suffix array (SA) provides a lexicographically ordered list of all the suffixes of a string. If $\mathrm{SA}[i]=j$, it means that the i-th smallest suffix of T is T_j, the suffix starting at position j in T. A related structure, the `LCP` array contains the length of the longest common prefixes between adjacent positions in the SA. Combining the SA with the `LCP` (Longest Common Prefix) information provides a powerful data structure for pattern matching. Worst-case linear-time direct SA construction algorithms are described in Adjeroh and Nan (2010), Kärkkäinen *et al.* (2006), Nong *et al.* (2009) and Puglisi *et al.* (2007). A linear-time algorithm for constructing ST from SA is presented in Adjeroh *et al.* (2008). Space-efficient constructions for representing the ST based on the SA have also been studied. Examples here include the enhanced suffix array (ESA; Abouelhoda *et al.*, 2004; Homann *et al.*, 2009), linearized suffix trees (LSTs; Kim *et al.*, 2008) and the virtual suffix tree (VST; Lin *et al.*, 2009). Compressed index representations, such as compressed STs and compressed SAs have also been studied (Grossi and Vitter, 2005; Sadakane, 2007; Välimäki *et al.*, 2009). Much earlier, Kurtz (1999) proposed direct methods for space-efficient construction of the ST, without using SAs. See Section 4.4 for further discussion on efficient ST representations. In this article, we focus on the use of standard SA as the basis for the proposed probabilistic SAs.

2.2 Variable length Markov models

A Markov model is a sequence of stochastic events $\{X_n, n=0,1,2,3,...\}$ with a state space that satisfies the Markov property:

$$\begin{aligned}P(X_{n+1}=j|X_n=i,X_{n-1}=i_{n-1},...,X_0=i_0)\\ =P(X_{n+1}=j|X_n=i,X_{n-1}=i_{n-1},...,X_{n-L}=i_{n-L})\end{aligned}$$

where, L is the order of the model. Thus, in an order-L Markov model, the current state is dependent on the past L states. Fixed length Markov models (FLMMs) represent a probabilistic finite state machine which can be used to model arbitrarily complex sequences. Such models learn the probability $P(\sigma|C)$, the conditional probability of a symbol σ, given its contexts C, where $\sigma\in\Sigma$ and $C\in\Sigma^L$, L is the order or memory length of the model, and is fixed. The FLMM of order L is represented as a $\Sigma^L\times\Sigma$ matrix. The space needed is thus in $O(\Sigma^{L+1})$.

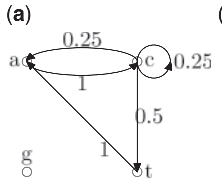
Variable length Markov models (VLMMs) differ from FLMM in an important way. VLMMs attempt to learn the conditional distribution of a symbol where the context length or model order could be varying, depending on the data being modeled. Essentially, for VLMM, $C\in\bigcup_{i=1}^{L}\Sigma^i$. This property of varying memory length implies that with the VLMM, Markov dependencies of varying order in the training data—both large and small—could be captured with ease. This flexibility of the VLMM, however, comes at a huge cost in terms of space requirement. To represent a VLMM of order L, we have to store L matrices, one for each order, from 1 to L. The total space requirement will be in $O(\Sigma^2+\Sigma^3+\cdots+\Sigma^{L+1})$ or $O(\Sigma^{L+2})$. The space is huge, even when L is small. Thus, the space requirement for Markov models is exponential in L, whether we consider fixed length or variable length models.

An important observation that could point to a potential reduction in the space requirement for Markov models is that for a given sequence of length N, there are $N(N+1)/2$ possible substrings in the sequence. (The number of unique substrings is typically smaller, and also depends on $L,|\Sigma|$ and the specific sequence.) Thus, there are at most $N(N+1)/2$ states that need to be represented in the Markov model for the sequence, for any given order. For the length-N sequence, the maximum order of a Markov model will be $N-1$. Thus, given the sequence, we can have a limit on the possible number of states in its Markov model.

2.3 Probabilistic suffix tree

The PST is based on the traditional suffix tree. Like the suffix tree, the PST represents all the $N(N+1)/2$ substrings from the root to the leaf

(a)



(b)

Probability	a	c	g	t
a	0	1	0	0
c	$\frac{1}{4}$	$\frac{1}{4}$	0	$\frac{1}{2}$
g	0	0	0	0
t	1	0	0	0

Fig. 1. (a) State diagram and (b) transition matrix for a first-order Markov model for an example sequence T=`accactact$`.

nodes. The PST models VLMMs, which means that the string depth is not fixed for every node. For an FLMM, its corresponding PST can be obtained from the PST of the VLMM by constraining each leaf node to be of the same string depth. The transition probability of a symbol on a given path is computed as the relative frequency of the symbol in the observed data, given the preceding substring on the path. The length of the substring used to determine such conditional probabilities is simply given by the order of the model.

Consider the sample sequence T=`accactact$`. Figure 1 shows its first-order transition matrix and the corresponding state diagram. Figure 2 shows the suffix tree and the corresponding PST. The PST is shown for the case of order $L=3$. In this PST, we label the transition probabilities for each symbol. The transition probability for a given symbol is calculated using the conditioning context.

The original PST algorithm (Ron *et al.*, 1996) needed an $O(LN^2)$ time to construct and prune the PST from a suffix tree. The improved algorithm (Mazeroff *et al.*, 2008) used balanced red–black trees (Cormen *et al.*, 1990) to construct the PST in $O(LN\log N)$ time complexity. (Apostolico and Bejerano, 2000) presented an $O(N)$ time algorithm using suffix links and reverse suffix links. (Leonardi, 2006) proposed an improved variation of the PST, called sparse probabilistic suffix tree (SPST) for use in protein family prediction. The SPST differs from the PST in that the SPST models sparse Markov chains (SMCs), whereby the conditioning contexts are defined by sparse sequences, that is, subset of sequences of amino acids. In an SMC, some contexts can be grouped together to form an equivalent class. In SPST, this grouping is performed by considering the contexts with similar prefixes. This leads to a type of partitioning of the contexts, based on their common prefixes. In practice, the effect is essentially a form of pruning of the PST, whereby the node depths are not fixed, but could vary depending on the sequence. This variability in node depths makes the SPST different from the PST with fixed order or node depth L [e.g. $L=20$ in Bejerano and Yona (2001)]. Results reported in Leonardi (2006) showed that the SPST improved on PST prediction by ∼2–4%, while maintaining the same general time complexity.

2.4 Empirical probabilities via SAs

In Apostolico and Bejerano (2000); Ron *et al.* (1996), the conditional probabilities required for the PST were computed as relative counts, using empirical probabilities, based on symbol frequencies from the observed data. To compute the empirical probabilities, we use the notions of *term frequency* (TF) and *document frequency* (DF) as used in information retrieval. The TF is simply the number of times a given term (or substring in our case) occurred in a given sequence. There are $N(N+1)/2$ substrings in a sequence of size N. Using a naïve algorithm, we will need to compute TFs for all the $N(N+1)/2$ terms. However, with the suffix tree for this sequence, we have N leaf nodes and at most N internal nodes. These $2N$ nodes represent the $N(N+1)/2$ substrings in the sequence. Therefore, some multiple substrings at different positions in the sequence must have been represented by the same node. These substrings represented in the same node must have the same frequency count. Since the node labels are unique in the suffix tree, this means that the multiple substrings in the same node are essentially the same substring, that was repeated multiple times along the sequence. Thus, each unique substring forms a ‘group’, whereby group members essentially differ in terms of their starting locations in the string T. While there are $O(N^2)$ possible substrings in T, there are at most $O(N)$ unique substrings (or groups). Hence, we need to compute the TF for only the $2N$ unique substrings.

Yamamoto and Church (2001) presented a data structure, called the *interval array*, to represent the groups as nodes in the suffix tree using a SA. For example, the substring ‘`ac`’ occurred three times in the sample sequence shown in Figure 2a. This ‘`ac`’ group corresponds to the interval $<1,3>$ (Table 1). Using the interval array, their algorithm calculated all the required TFs in $O(N)$ time. Table 1 shows the interval array for the sample sequence T=`accactact$`. From the table, we can describe two important properties of the interval array (or substring groups).

(1) There are at most $2N$ groups in a sequence of size N. Substrings in the same group have the same statistics (for example: TF and DF) and the same derivative measurements.

(2) An lcp-delimited interval $<i,j>$ is constructed using the LCP array, where (i,j) is an interval on the SA. An lcp-delimited interval $<i,j>$ must meet the condition:

$\max(\text{LCP}[i],\text{LCP}[j+1]) < \text{Length}_{\text{group}(i,j)} \leq \min(\text{LCP}[i+1],\text{LCP}[i+2],...,\text{LCP}[j])$

where, $\text{Length}_{\text{group}(i,j)}$ is the LCP of the suffixes that belong to the same group with the $<i,j>$ interval. Let α and β be two substrings in the same lcp-delimited interval $<i,j>$. Then, $\text{TF}(\alpha)=\text{TF}(\beta)=j-i+1$.

3 METHOD—PSA

Previous work (Apostolico and Bejerano, 2000; Ron *et al.*, 1996) has represented Markov models and PFA using the PST. Here, we present a space-efficient data structure to represent such finite state machines. We call our data structure the PSA, since it is built on the SA data structure. The PSA uses an array of nodes to capture the branching structure in the suffix tree, and other auxiliary arrays to maintain information needed for learning from the observed data. Learning in the PSA is performed by computing conditional probabilities at each node in the PSA using empirical probabilities computed via the TF.

3.1 Data structure

Essentially, we use the probabilistic SA to simulate the PST. Each node in the PST has a corresponding position in the PSA. The basic PSA structure has three types of attributes. The first category of attributes is the *foundation attributes*, which consists of the original text and its SA. Construction of the SA is in $O(N)$ time and space, using any of the various linear-time, linear-space algorithms [see for example Adjeroh and Nan (2010); Kärkkäinen *et al.* (2006); Nong *et al.* (2009); Puglisi *et al.* (2007)].

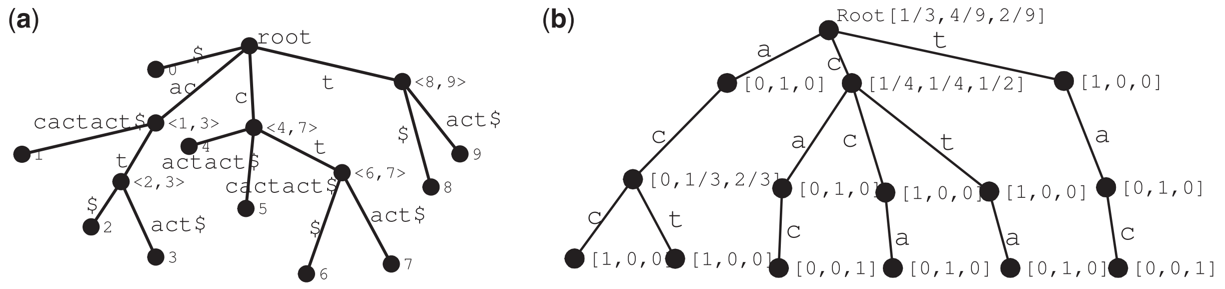


Fig. 2. Example suffix tree (**a**) and PST (**b**) for the string T=accactact\$. The trees are shown without the suffix links. The numbers at each node of the PST, say u, encode the conditional probabilities $P(\sigma|C)$ of observing the symbol σ after observing the context C, which in this case is the sequence corresponding to the node label, $LL(u)$. Angle brackets at the internal nodes in the ST denote the intervals in the SA. These are used by the PSA.

Table 1. Interval array for T=accactact\$

Interval	LCP	Term frequency
<2,3>	3	2
<1,3>	2	3
<6,7>	2	2
<4,7>	1	4
<8,9>	1	2

See Figure 2a for the position of each interval in the suffix tree.

The second category of attributes are the *internal node attributes*. These are derived from the interval array, which is determined following Yamamoto and Church (2001). These are used to represent the internal nodes in the suffix tree, including the suffix links. The suffix link is the link from an internal node to its corresponding suffix node. The third type of attributes are *measurement attributes*. These record measurement information, such as TF, DF and conditional probabilities, which are needed to compute probabilities in the Markov model. Table 2 shows the three types of attributes used in the PSA. The TF can be calculated at run time. Thus the attribute *Conditional Probability* (`cProb`) which is based on `TF` can be computed also at run time. We use the term *length of the PSA* (denoted by M) to refer to the number of nodes in the PSA. The term *length* emphasizes the fact that our PSA nodes are stored as arrays.

3.1.1 Internal node attributes. The internal node attributes are derived from the interval array. The pair <`Start`, `End`> represents the interval position of a node in the SA. The PSA internal node attributes are used to simulate the internal nodes in a suffix tree. The attribute `sLink` is a regular suffix link from the current internal node to its suffix node. We use this link to continue the searching process when a mismatch occurs at the time of prediction. The attribute `Parent` is the parent node of the current node. The internal node attributes including the suffix link are constructed using Algorithm BUILDPSA.

3.1.2 Measurement attributes. These attributes are dependent on the application. For example, for applications in document feature selection, or in protein sequence classification, we only compute the TFs and the conditional probabilities (as needed) at run time, using the formula (`End`−`Start`+1). In other applications, such as protein sequence clustering, we may need to compute DF for the

Table 2. PSA attributes

Attribute type	Attribute description	Notation	Data type
Foundation attributes	Original text	T	char
	Suffix array	SA	integer
Internal node attributes	Start	`Start`	integer
	End	`End`	integer
	Suffix link	`sLink`	integer
	Parent	`Parent`	integer
Measurement attributes	Term frequency	TF	integer[a]
	Document frequency	DF	integer
	Cond. probability ($P(S_t\|S_k,...,S_{t-1})$)	`cProb`	float[a]

[a]indicates an attribute that will be computed at run time, and hence does not need extra storage.

classes, rather than just the DF. The attributes are also dependent on the methods used in calculating the probabilities in the Markov model. Thus, we focus on the method for computing the conditional probabilities, and the probability of a node in the VLMM. For a given node with node label, say $(S_1,...,S_n)$, its probability, P_{VLMM} is given by:

$$P_{\mathrm{VLMM}}=P(S_1,...,S_n)=P(S_n|S_1,...,S_{n-1}),...,P(S_2|S_1)P(S_1) \quad (1)$$

If $S_1,...,S_{t-1}S_t$ (with $1\le t\le N$) does not occur in the training data, we find the longest suffix of $S_1,...,S_{t-1}S_t$ which occurred in the training data. Assume $S_k,...,S_{t-1}S_t$ $(1\le k\le t)$ is the longest suffix of $S_1,...,S_{t-1}S_t$ that occurred, then $P(S_t|S_1,...,S_{t-1})=P(S_t|S_k,...,S_{t-1})$. Thus,

$$P(S_t|S_1,...,S_{t-1})=P(S_t|S_k,...,S_{t-1})=\begin{cases}\dfrac{\mathrm{TF}_{S_t}}{N} & :k=t\\[2ex] \dfrac{\mathrm{TF}_{S_k,...,S_t}}{\mathrm{TF}_{S_k,...,S_{t-1}}} & :k<t\end{cases} \quad (2)$$

Here, TF_u is the term frequency at the node with node label u. We make two observations: *First*, if the terminal symbol S_t of a path $S_k,...,S_{t-1}S_t$ is a first symbol in an edge of the suffix tree, then the conditional probability $P(S_t|S_k,...,S_{t-1})$ is calculated by the frequency of the current node divided by the frequency of the parent node. *Second*, if the terminal symbol S_t of a path $S_k,...,S_{t-1}S_t$ is not a first symbol in an edge of the suffix tree, then the conditional

Table 3. Example PSA internal nodes for $T=$`accactact$`

PSA index	Interval	Suffix link	Probability expression	Conditional probability
1	$<2,3>$	2	$P(t\|ac)$	$\frac{2}{3}$
2	$<1,3>$	3	$P(a)$	$\frac{1}{3}$
2	$<1,3>$	3	$P(c\|a)$	1
3	$<6,7>$	4	$P(t\|c)$	$\frac{1}{2}$
4	$<4,7>$	-1	$P(c)$	$\frac{4}{9}$
5	$<8,9>$	-1	$P(t)$	$\frac{2}{9}$

See Figure 2a for the position of each PSA node (i.e. each interval) in the suffix tree.

Table 4. Example PSA leaf nodes for the sequence $T=$`accactact$`

SA index	Probability expression	Conditional probability
1	$P(c\|ac)$	$\frac{1}{3}$
3	$P(a\|act)$	1
4	$P(a\|c)$	$\frac{1}{4}$
5	$P(c\|c)$	$\frac{1}{4}$
7	$P(a\|ct)$	$\frac{1}{2}$
9	$P(a\|t)$	1

probability $P(S_t|S_k,...,S_{t-1})$ is 1. We call this a *trivial conditional probability*. Thus, we need to calculate the conditional probabilities for only the first symbol in each edge. When we determine that the terminal symbol of path is not a first symbol in an edge, we simply return 1 for the conditional probability of the symbol.

3.1.3 Example PSA. Tables 3 and 4 show the nature of the PSA nodes and the order-3 conditional probabilities in a PSA, using the sequence $T=$`accactact$` used in Figure 2. The entries in Table 3 are directly calculated from the interval array. We notice that $P($`c`$|$`a`$)$ is 1. This indicates the substring '`ac`' is represented on one edge and that the terminal symbol '`c`' is the second symbol on this edge. This is a trivial conditional probability.

Entries in Table 4 are computed from the PSA leaf nodes. We only showed the non-trivial conditional probabilities on the leaf nodes. These conditional probabilities are calculated based on the first observation described in Section 3.1.2. The numerator is 1 since this is a leaf node which must have a frequency of 1. The denominator is the frequency of the substring corresponding to the node label of the parent of the current leaf node. This is easily obtained as (`End`$-$`Start`$+1$) using the elements in the interval array. The PSA nodes can be compared with the example PST shown in Figure 2.

3.2 Constructing the PSA

Having described the building blocks for the PSA, we are now ready to describe how we put them together to construct the PSA. For a given input sequence, algorithm BUILDPSA uses five major steps to construct the PSA data structure. The first step is the construction of the SA from the original sequence. We use standard linear-time, linear-space algorithms for this step. The next two steps construct the tree-like structure used in the PSA. Step 2 constructs the interval array (*iArray*) using the `print_LDIs_stack` function defined in Yamamoto and Church (2001). The next step maps each position in the input sequence to its interval. This step calculates the parent node of each node. The final step computes the inverse SA. Using the inverse SA the algorithm determines a link that allows the interval array to point to the next position, based on which the the suffix link for each PSA node is determined.

A more detailed description of the PSA construction steps and their computational complexity analysis can be found in Lin (2011).

Build probabilistic SA

```
BUILDPSA($T$)
1  SA ← BUILDSA($T$)
2  $iArray$ ← print_LDIs_stack(SA)
3  <$posIntv$, $parent$> ← BUILDINTERVALTREE(SA, $iArray$)
4  PSA ← BUILDSUFFIXLINK($posIntv$, $parent$)
```

3.3 Prediction with VLMM via the PSA

An important procedure in Markov models is to compute the probability that a given test pattern is generated by the model. This is particularly important in our problem of modeling protein families using VLMMs. For instance, given a protein family with some known sequences, one may be interested in checking if a given unknown protein sequence belongs to the family. This can be done by first constructing a model of the family (i.e. the PSA in our case) using the known sequences. Given a protein family with Z sequences, say $T_1,T_2,...,T_Z$, this is performed by constructing the PSA for the concatenated sequence: $T=T_1\$_1T_2\$_2...T_Z\$_Z$. To determine whether the unknown protein sequence belongs to the family, we use the model to compute the probability that the unknown sequence is generated by the model.

For the VLMM, we denote this probability as the P_{VLMM} of the input pattern. Algorithm VLMM-PREDICTION calculates the P_{VLMM} of a test sequence using the PSA data structure. This algorithm uses Equation (1) to compute $P(S_1),P(S_2|S_1),...$ by searching for the subpatterns $S_1,S_1S_2,...$ When there is a mismatch while searching with the subpattern, say $S_k,...,S_t$, the algorithm jumps to the node pointed to by the suffix link attribute of the current node. Thus, matching proceeds from the node representing the suffix $S_{k+1},...,S_t$, after accounting for the prefix of this suffix, which has already been matched in the previous step. When a mismatch occurs, the algorithm uses the previously computed index to determine the suffix link. Thus the search will be redirected to the new branch following the suffix link. The algorithm now uses the longest suffix of the subpattern that so far matched in the previous matching step as the new subpattern, and restarts matching from the symbol that mismatched. This redirection is a constant time operation.

The foregoing implies that, given a sequence $T=T[1,...,N]$, with symbols from an alphabet Σ, and the PSA for T, we can decide on whether a pattern $P=P[1,...,m]$ is generated by the same variable length Markov chain that generated T in $O(m\log\frac{N}{|\Sigma|})$ time.

We can use the predicted probability above to perform protein sequence classification. Suppose we have F protein families and we have computed the PSA for each family. Let PSA_k be the model constructed using the k-th protein family. Further, let

$P_{\text{VLMM}}(P, \text{PSA}_k)$ be the probability that protein sequence P is generated by PSA_k, as returned by algorithm VLMM-PREDICTION. Then, we classify P to protein family f, where,

$$f = \underset{k=1,\dots,F}{\text{argmax}} \{P_{\text{VLMM}}(P, \text{PSA}_k)\}. \qquad (3)$$

Prediction with VLMM via the PSA

```
VLMM-PREDICTION($P$, PSA)
1  $s \leftarrow 1$, $Prob \leftarrow 1$, $L \leftarrow 1$, $R \leftarrow N$, $index \leftarrow 0$, $m \leftarrow |P|$
2  for ($i \leftarrow 1$ to $m$) do
3    Search $P[s...i]$ in PSA.SA with parameter $L, R$
4    if mismatch do
5      $index \leftarrow$ PSA.$index$.sLink
6      $L \leftarrow$ PSA.$index$.Start, $R \leftarrow$ PSA.$index$.End
7      $s \leftarrow s+1$
8      if $i \leq s-1$ do
9        $i \leftarrow i+1$
10     end if
11   else
12     $leftPosn \leftarrow$ LMATCHEDPOSN($P[s...i]$,PSA.SA,$L$,$R$)
13     $index \leftarrow$ SearchPSA(PSA,$leftPosn$,$i-s+1$)
14     $Prob \leftarrow Prob$ * PSA.$index$.cProb
15     $L \leftarrow$PSA.index.Start, $R \leftarrow$ PSA.index.End
16   end if
17 end for
18 return $Prob$
```

4 RESULTS AND DISCUSSION

We performed experiments on protein sequences to test the proposed PSA data structure. The experiments were performed using a DELL PC, with 4 × 2.67 GHz CPU, and 8G memory, running Ubuntu 10.10 Linux operating system. All programs were compiled using *gcc*.

4.1 Predicting protein families

Gapped-BLAST (Altschul *et al.*, 1997) provides a simple but popular approach to protein family prediction. Perhaps the most popular and accurate methods for protein family classification are based on profile HMMs, introduced in the 1990's (Durbin *et al.*, 1998; Eddy, 2010). HMM is more accurate than BLAST, and also performs better in detecting distant homologs. In (Bejerano and Yona, 2001), the PST was compared with both Gapped-BLAST and an earlier implementation of HMM, using the old Pfam 1.0 dataset (Bateman *et al.*, 2004). In terms of prediction accuracy, the PST generally did better than BLAST, but under-performed HMM by ∼2–5% (90.7% versus 91.5–96.1%). The PST was, however, much faster than HMM for both training and searching. In Leonardi (2006), the SPST was introduced as an improved variant of the PST, resulting in ∼2% improvement in accuracy. The latest implementation of profile HMM is HMMER 3.0 `http://hmmer.janelia.org/` (Eddy, 2010) which is now essentially as fast as BLAST, but more accurate.

Our major objective was to develop a time- and space-efficient alternative to the PST/SPST. To test the performance of PSA in modeling protein sequences and in predicting the family for unknown sequences, we downloaded the current Pfam database (release 25.0; Bateman *et al.*, 2004; Finn *et al.*, 2008). We use family members from the unaligned sequences to generate the PSA, one PSA for each family. We selected the first 48 families with 12 or more members (based on alphabetical ranking of the family names). For each family, we selected the first 1000 protein sequences (or all sequences if family size <1000) to make up our dataset. The result is a total of 46 810 protein sequences from the 48 families. For each family, we selected 80% of the protein sequences (37 448) for training, used the remaining 20% (9362 sequences) for testing.

Table 5. Summary performance in protein family classification using the PSA, SPST and HMMER3

	Size	# MD (PSA)	TP (PSA)	# MD (HMM)	TP (HMM)	# MD (SPST)	TP (SPST)
Mean	195	174.63	89.56%	180.35	92.55%	175.25	89.82%
STD	20	28.91	0.11	34.83	0.15	25.85	0.09
Max	200	200.00	100%	200.00	100%	200.00	100%
Min	103	90.00	51.50%	59.00	29.50%	88.00	53.00%
Total	9362	8382		8657		8457	

Results are for the first 48 protein families (listed alphabetically) with 12 or more members in the Pfam 25.0 database. (TP denotes true positive, MD denotes missed detection).

Below, we compare the proposed PSA with SPST (best performing variant of PST), and HMMER 3.0 (latest implementation of profile HMM).

For HMMER3, we use the e-value to decide to which family a given test sequence should be assigned. Thus, in some cases HMMER3 returns a 'no match', which we consider as a missed detection. To use the PSA, for each sequence in the test dataset, we compute the P_{VLMM} using the PSA structure for each family. We then assign the protein sequence to the family with the maximum probability. When the maximum probability is obtained using the model of the correct family (whose PSA is generated without the test sequence), we say we have a correct classification (true positive), otherwise, there is a classification error. This simple approach avoids the difficult problem of setting thresholds for correct classification.

Table 5 shows the summary classification performance on the Pfam dataset using the PSA. Detailed classification results, showing performance on each family are provided as Supplementary Table S1. For comparison, we have also included the results obtained using the SPST and HMMER3 on the same dataset. As expected, both SPST and PSA produced comparable results with respect to modeling and classification of protein families. The SPST had an average true positive rate of 89.82%, whereas the PSA had 89.56%. These were just ∼3% lower than the average performance (namely 92.55%) obtained using HMMER 3.0. We performed a harder test, by using 20/80 data partitioning (rather than 80/20) for training versus testing. The overall true positive rate was reduced for each of the three schemes (PSA 85.40%, SPST 84.49% and HMMER3 89.91%; see Supplementary Table S2). The results, however, show that with respect to prediction accuracy, SPST and PSA are not significantly worse than HMMER3, the current state-of-the-art, even under the more challenging scenario of 20/80 split for training versus testing.

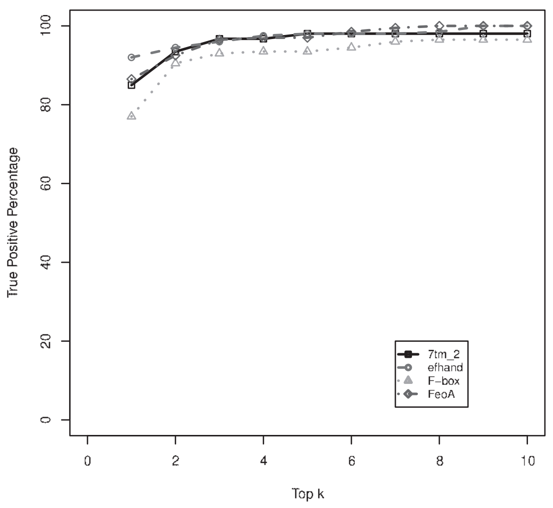


Fig. 3. Top-k classification rate for sample protein families using the PSA.

To evaluate the impact of increasing database set size and diversity on the PSA performance, we selected 1000 protein families with 1000 or more members in Pfam. For each family, we selected the first 1000 protein sequences to make up our dataset. The result is a total of 1 000 000 protein sequences from the 1000 families. For each family, we selected 80% of the protein sequences (800 000) for training, used the remaining 20% (200 000 sequences) for testing. We compare the performance of the proposed PSA with that of HMMER3 on the large dataset. The performance decreased somewhat, for both HMMER3 and PSA. The performance for PSA changed from 89.56% (for $F=48$ families) to 87.10% (for $F=1000$ families), or a decrease of 2.46%. However, HMMER3 showed a less performance decrease of ∼1% [from 92.55% ($F=48$) to 91.58% ($F=1000$)]. Detailed comparative results, showing the classification performance on each family are provided as Supplementary Table S4.

One advantage of performing classification using Equation (3), is that, when there is an error in the classification, we can consider the family with the next highest predicted probability. For instance, we can consider the top-k classification rate, which shows the probability of finding the correct protein family within the first k families, as ordered based on their generated probabilities, using the test sequence. Figure 3 shows the classification performance of the PSA on some sample families, using the top-k classification rate. We can observe how the classification performance rapidly approaches 100% after the first few k-values.

4.2 Space consideration

A major problem with suffix trees is their practical memory space requirement. Although, they have the same theoretical linear space requirement as SAs, in practice, suffix trees consume much more space (Adjeroh *et al.*, 2008; Gusfield, 1997; Puglisi *et al.*, 2007). This was our primary motivation for developing the PSA. Table 6 shows the summary data on the protein families in Pfam 25.0 used in our experiments. Table 7 compares the memory space required to construct the SPST and HMM with that required for the PSA.

First, we can observe the nature of the protein sequences (Table 6). The maximum branching factor ($\gamma=\frac{M}{N}$) observed was 0.687, whereas the minimum was 0.432. The average was 0.538. The branching factor provides an indication of the nature of the sequence. Higher values typically indicate less repetition in the sequence. As a key performance measure, we used the *memory consumption factor* (MCF), defined as the ratio of the required memory to the total sequence length (N) of the family. We compared the MCF for PSA, SPST and HMMER3. Table 7 shows that the PSA ratio was steady at about ($21.58\pm6.32N$) bytes. The maximum memory required by the PSA for any of the families was $42.11N$ bytes. This can be compared with the (mean and maximum) memory needed for SPST ($27.55\pm13.16N, 63.01N$) and HMMER3 ($47\pm24.95N, 140.3N$). Supplementary Figure S1 shows more detailed information on the memory consumption needed to construct the data structures, using the Pfam 25.0 protein families. We can observe that the minimum MCF for HMMER3 was larger that the average MCF for PSA. Table 8 shows the corresponding MCF for the three schemes during the time of prediction (testing).

Perhaps, more significantly, while the PSA memory is relatively constant independent of the sequence or family, we can observe the significant fluctuation in the memory needed for the SPST and HMMER3, as captured by the range and SD on the MCF. This relative stability can be observed during both PSA construction and PSA prediction. For the dataset, the PSA required an average MFC of ($13.6N\pm0.96$) and a maximum MFC of $16N$ for storage, after construction.

Table 6. Summary data on the first 48 families in Pfam

	Family size	Length (N)	No. of internal nodes (M)	$\gamma=\frac{M}{N}$	Max LCP	Mean LCP
Mean	780	125 853.15	68 811.98	0.54	117.90	7.31
STD	80	70 422.77	40 160.10	0.06	53.50	3.12
Max	800	319 691.00	174 657.00	0.69	250.00	19.00
Min	412	23 879.00	12 673.00	0.43	42.00	4.00

Table 7. Construction memory needed for the PSA, SPST and HMMER3

	PSA (KB)	MCF	HMM (KB)	MCF	SPST (KB)	MCF
Mean	2574.58	21.58	4626.83	47.00	2630.33	27.55
STD	1501.54	6.32	1476.17	24.95	480.51	13.16
Max	5712.00	42.11	9604.00	140.31	3812	63.01
Min	868.00	15.41	3272.00	23.53	1456	11.38

Table 8. Memory consumption for PSA, SPST and HMMER3 during predition

	PSA (KB)	MCF	HMM (KB)	MCF	SPST (KB)	MCF
Mean	2234.25	19.90	4776.92	45.57	2043.92	19.94
STD	968.57	4.18	2102.45	19.25	796.73	13.11
Max	4712.00	36.54	13 028.00	107.04	3104.00	60.90
Min	852.00	14.72	2488.00	18.53	60.00	0.37

Table 9. Construction time comparison for PSA, SPST and HMMER3

	PSA	HMMER3	Speed-up	SPST	Speed-up
Mean	0.08	0.90	10.80	21.11	255
STD	0.06	0.51		14.48	
Max	0.26	2.28		70.38	
Min	0.01	0.11		2.28	

Recorded time is time needed per family (in seconds). Speedup is computed as the ratio with respect to PSA time.

Table 10. Prediction time comparison between PSA, SPST and HMMER3

	PSA	HMMER3	Speed-up	SPST	Speed-up
Mean	1.56	1.30	0.84	577.35	371.14
STD	0.47	1.32		118.96	
Max	2.67	7.35		823.94	
Min	0.84	0.21		262.79	

Recorded time (in seconds) is prediction time per family—i.e. total time needed to predict all members in the family against all the other families.

4.3 Computational time requirement

Table 9 shows the summary of the time required for constructing the PSA and the PST data structures. Table 10 shows the corresponding summary of the time needed for prediction using the models. The tables show that for prediction, on average, the PSA was ∼10.8 times faster to build than HMMER3, and ∼255 times faster than constructing SPST. For prediction, the PSA was ∼371 times faster than SPST.

4.4 Discussion

The proposed data structure is based on the standard SA, and hence can be implemented using compressed index structures such as compressed suffix arrays (CSAs; Grossi and Vitter, 2005; Sadakane, 2007; Välimäki *et al.*, 2009). Grossi and Vitter (2005) showed that for a general alphabet, Σ, with $|\Sigma|>2$, the CSA can be constructed in $O(N\log_{|\Sigma|}N)$ processing time and stored in $\left((1+\frac{1}{2}\log\log_{|\Sigma|}N)N\log|\Sigma|+5N+O\left(\frac{N}{\log\log N}\right)\right)$ bits, such that each *lookup* operation can be performed in $O\left(\log\log_{|\Sigma|}N\right)$ time. Similarly, in theory, the PST or SPST can be implemented using compressed suffix trees (CSTs). (Sadakane, 2007) showed that a CST with full functionality (including suffix links) can be constructed in $O(N)$ time, and represented using $O(N\log|\Sigma|)$ bits of storage space. He showed that the CST can be implemented using $(NH_h+6N+o(N))$ bits, where H_h is the order-h entropy of the original sequence. For the CSA, the storage requirement can be reduced to $(NH_h+O\left(\frac{N\log\log N}{\log_{|\Sigma|}N}\right)$ bits. These will no doubt result in a lower memory requirement when compared with our current result using standard SAs. However, apart from the usual slowdown in processing speed using CSAs or CSTs, some types of traversals and new attributes required for the PSA (or PST/SPST) could be quite difficult to implement on a standard CSA (or CST). This type of improvement will be an interesting direction for further work on space-efficient probabilistic suffix structures. The proposed method

Table 11. Summary and comparative performance in protein family prediction using the improved PSA

	PSA	PSAw(avePr)	PSAw(maxPr)	HMMER3	SPST
Mean	89.56%	86.80%	91.76%	92.55%	89.82%
STD	0.11	0.15	0.10	0.15	0.09
Max	100%	100%	100%	100%	100%
Min	51.50%	43.00%	69.50%	29.50%	53.00%

Indicated results are true positive rates in percentage (%). For PSA, $w=10$ using average probability, and $w=80$ using maximum probability.

is not directly affected by the computer word length (32 bit or 64 bit), so far as the data structure including input data can fit into main memory. For very large datasets where this may not be the case, methods will need to be considered for more efficient I/O operations (Ferragina, 2010).

In terms of prediction accuracy, a potential approach to improve the performance will be to consider prediction based on short fragments of the test sequence. These will provide more locality in the analysis, and hence may be able to improve the ability for detecting distant homology. One way to use the predictions from different fragments will be to choose the one with the family with the maximum predicted probability over all the fragments as the predicted family. However, other approaches are also possible, for instance by combining the predictions using some well defined protocol. We tested this approach, using overlapping windows of various sizes, w. The best results were obtained at $w=10$ (86.80%) using average of window probabilities, and at $w=80$ (91.76%) using the maximum probability. Table 11 provides a summary of the results, showing that with the improvement, the average PSA prediction accuracy is now <0.8% below that of HMMER3. Detailed results are in Supplementary Table S1.

Further, as can be seen from the results (Supplementary Tables S1 and S2), there are cases where the PSA performed signifcantly better than HMM, e.g. by as much as 40% for *FAD_binding_3* or 59% (for *FA_hydroxylase*). Thus, a potential improvement could be obtained by combining prediction results from the PSA with those from non-PST-based methods, such as the profile HMM, for instance, using an approriate classifier fusion scheme (Kittler *et al.*, 1998).

We acknowledge some basic problems with using VLMMs in general (whether PST, SPST or PSA). One problem is that these tend to provide a global characterization of the sequence, and hence may have problems when similarity between sequences is localized within a smaller region of the proteins, or is interspersed between islands of non-similar regions. This will thus affect the ability to detect distant homology, or the ability to handle a previously unseen mutation in a protein sequence that is being analyzed. This last problem is akin to the *zero-frequency-problem*, and various methods have been proposed, for instance, using a default probability distribution, or a Poisson process model (Witten and Bell, 1997). A related problem with the current PSA implementation is that the conditioning context for the symbol probability reverts to the longest observed suffix, meaning that the conditional probability would be based on a small number of counts. This may not always correspond to the true probability. One solution would be to use some kind of interpolation scheme, in which the probabilities are estimated based on shorter prefixes, for instance, bigrams or trigrams. Though these may still not always provide the true probabilities, the estimates are likely to be closer to the true values, and also likely to be more robust. This approach is somewhat similar to the successive reduction in context size as used in context models such as prediction by partial matching (PPM) (Cleary and Teahan, 1997; Cleary and Witten, 1984; Zhang and Adjeroh, 2008) used in effective data compression. Another approach is to use approximate matching in evaluating the similarity between prefixes, rather than exact matching as is used in the current implementation. (Lin, 2011) provides some more discussions on improvements on the prediction strategy.

We mention one issue with the current implementation of protein family prediction using the PSA. Currently, the family with the highest probability as reported by $P_{\mathrm{VLMM}}()$ is *always* chosen as the predicted family [see Equation (3)]. This means that, there is no significance attached to the prediction, and that the system must report a prediction for any given test sequence. Thus, when the test sequence does not belong to any of the families in the training set, the assignment must necessarily be in error. This problem can be reduced by including a mechanism that can return a ‘no match’ (similar to HMMER3), for instance, based on some measure of confidence on the prediction. Using the PSA approach, one simple way to do this will be to consider the maximum and second maximum probabilities returned by $P_{\mathrm{VLMM}}()$. If the difference between the two is above a certain threshold, the system will report a prediction (i.e. family with the maximum probability). Otherwise, it reports a ‘no match’.

5 CONCLUSION

We have presented the PSA, a data structure for representing information in VLMMs. The PSA provides the same functionality as the PST, but at a significantly reduced time and space requirement. Given a sequence of length N, construction and learning in the PSA is done in $O(N)$ time and $O(N)$ space, independent of the Markov order. Prediction using the PSA is performed in $O\left(m\log\frac{N}{|\Sigma|}\right)$ time, where m is the pattern length, and Σ is the symbol alphabet.

We have shown practical results on the comparative performance of PSA, SPST and HMMER3 using protein families from Pfam 25.0. In terms of modeling and prediction, SPST and PSA produce equivalent results (SPST 89.82%, PSA 89.56%). These were slightly lower than the 92.55% obtained using HMMER3 on the same dataset. An improved version of PSA prediction (predicting families using fragments from the test sequence) improved PSA perfomance to 91.7%, thus reducing the performance gap with HMMER3 to just 0.79%.

The average practical construction space for the protein families tested was $21.58\pm6.32N$ bytes using the PSA, $27.55\pm13.16N$ bytes using SPST and $47\pm24.95N$ bytes for HMMER3. The maximum practical space needed was $42.11N$ for PSA, $63.01N$ for SPST and $140.3N$ for HMMER3. With respect to construction time, the PSA was 255 times faster than SPST, and 11 times faster than HMMER3. Our results show that, the PSA provides an accuracy similar to that of PST/SPST and HMMER3 in protein family prediction, but at a significantly lower time and space requirement.

ACKNOWLEDGEMENTS

We thank Dr Gill Bejerano for the help with the PST. We also thank the reviewers for their critique and suggestions which have no doubt led to a significant improvement in the quality of the article.

Funding: DOE CAREER grant no.: DE-FG02-02ER25541 and WV-EPSCoR RCG grant.

Conflict of Interest: none declared.

REFERENCES

Abe,N. and Warmuth,M. (1992) On the computational complexity of approximating distributions by probabilistic automata. *Mach. Learn.*, **9**, 205–260.

Abouelhoda,M.I. *et al.* (2004) Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, **2**, 53–86.

Adjeroh,D. and Nan,F. (2010) Suffix sorting via Shannon-Fano-Elias codes. *Algorithms*, **3**, 145–167.

Adjeroh,D. *et al.* (2008) *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*. Springer, New York.

Altschul,S. *et al.* (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Apostolico,A. and Bejerano,G. (2000) Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. *J. Comput. Biol.*, **7**, 381–393.

Bateman,A. *et al.* (2004) The Pfam protein families database. *Nucleic Acids Res.*, **32**, D138–D141.

Begleiter,R. *et al.* (2004) On prediction using variable order Markov models. *J. Artif. Intell. Res. (JAIR)*, **22**, 385–421.

Bejerano,G. and Yona,G. (2001) Variations on probabilistic suffix trees: statistical modeling and prediction of protein families. *Bioinformatics*, **17**, 23–43.

Cleary,J.G. and Teahan,W.J. (1997) Unbounded length contexts for ppm. *Comput. J.*, **40**, 67–75.

Cleary,J.G. and Witten,I.H. (1984) Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, **COM-32**, 396–402.

Cormen,T.H. *et al.* (1990) *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Durbin,R. *et al.* (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK.

Eddy,S.R. (2010). Hmmer user’s guide: Biological sequence analysis using profile hidden markov models. In *HMMER User’s Guide*. http://hmmer.janelia.org/

Ephraim,Y. *et al.* (1989) A minimum discrimination information approach for hidden Markov modeling. *IEEE Trans. Inf. Theory*, **35**, 1001–1013.

Ferragina,P. (2010) Data Structures: Time, I/Os, Entropy, Joules! In de Berg,M. and Meyer,U. (eds) *Algorithms – ESA 2010.* Vol. 6347 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, pp. 1–16.

Finn,R.D. *et al.* (2008) The Pfam protein families database. *Nucleic Acids Res.*, **36**, 281–288.

Gillman,D. and Sipser,M. (1994) Inference and minimization of hidden Markov chains. In *COLT Proceedings of the Seventh Annual Conference on Computational Learning Theory*, New York, NY, USA, , pp. 147–158.

Grossi and Vitter (2005) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, **35**, 378–407.

Gusfield,D. (1997) *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK.

Homann,R. *et al.* (2009) mkESA: enhanced suffix array construction tool. *Bioinformatics*, **25**, 1084–1085.

Kärkkäinen,J. *et al.* (2006) Linear work suffix array construction. *J. ACM*, **53**, 918–936.

Kim,D.K. *et al.* (2008) Linearized suffix tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays. *Algorithmica*, **52**, 350–377.

Kittler,J. *et al.* (1998) On combining classifiers. *IEEE Trans. Pattern Anal. Mach. Intell.*, **20**, 226–239.

Kurtz,S. (1999) Reducing the space requirement of suffix trees. *Softw. Prac. Exp.*, **29**, 1149–1171.

Leonardi,F.G. (2006) A generalization of the PST algorithm: modeling the sparse nature of protein sequences. *Bioinformatics*, **22**, 1302–1307.

Lin,J. *et al.* (2009) The virtual suffix tree. *Int. J. Found. Comput. Sci.*, **20**, 1109–1133.

Lin,J. (2011) Suffix structures and circular pattern problems. PhD Thesis, West Virginia University.

Mazeroff,G. *et al.* (2008) Probabilistic suffix models for API sequence analysis of windows XP applications. *Pattern Recogn.*, **41**, 90–101.

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

Nong,G. *et al*. (2009) Linear time suffix array construction using D-critical substrings. In Kucherov,G. and Ukkonen,E. (eds)*CPM*. Vol. 5577 of *Lecture Notes in Computer Science*, Springer, pp. 54–67.

Puglisi,S.J. *et al*. (2007) A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, **39**.

Ron,D. *et al*. (1996) The power of amnesia: learning probabilistic automata with variable memory length. *Mach. Learn.*, **25**, 117–149.

Sadakane,K. (2007) Compressed suffix trees with full functionality. *Theory Comput. Syst.*, **41**, 589–607.

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

Välimäki,N. *et al*. (2009) Engineering a compressed suffix tree implementation. *ACM J. Exp. Algor.*, **14**.

Witten,I.H. and Bell,T.C. (1997) The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inform. Theory*, **37**, 1085–1094.

Yamamoto,M. and Church,K.W. (2001) Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Comput. Linguist.*, **27**, 1–30.

Zhang,Y. and Adjeroh,D. (2008) PPAM: prediction by partial approximate matching for lossless image compression. *IEEE Trans. Image Process.*, **17**, 924–935.