

Fast and Adaptive Variable Order Markov Chain Construction

Marcel H. Schulz^{1,3}, David Weese², Tobias Rausch^{2,3}, Andreas Döring², Knut Reinert², Martin Vingron¹

¹ Department of Computational Molecular Biology, Max Planck Institute for Molecular Genetics, Ihnestr. 73, 14195 Berlin, Germany,

{marcel.schulz,martin.vingron}@molgen.mpg.de

² Department of Computer Science, Free University of Berlin, Takustr. 9, 14195 Berlin, Germany,

{weese,rausch,doering,reinert}@inf.fu-berlin.de

³ International Max Planck Research School for Computational Biology and Scientific Computing

Abstract. Variable order Markov chains (VOMCs) are a flexible class of models that extend the well-known Markov chains. They have been applied to a variety of problems in computational biology, e.g. protein family classification. A linear time and space construction algorithm has been published in 2000 by Apostolico and Bejerano. However, neither a report of the actual running time nor an implementation of it have been published since. In this paper we use the lazy suffix tree and the enhanced suffix array to improve upon the algorithm of Apostolico and Bejerano. We introduce a new software which is orders of magnitude faster than current tools for building VOMCs, and is suitable for large scale sequence analysis.

1 Introduction

Markov chains are often used in computational biology to learn representative models for sequences, as they can capture short-term dependencies exhibited in the data. The fixed order L of a Markov chain determines the length of the preceding context, i.e. the number of dependent positions, which is taken into account to predict the next symbol. A severe drawback of fixed order Markov chains is that the number of free parameters grows exponentially with L , s.t. training higher order Markov models becomes noisy due to overfitting. These considerations have led to the proposal of a structurally richer class of models called variable order Markov chains, which can vary their context length. There are two prominent methods in use by the community. One is the tree structured context algorithm of Rissanen [1], and the other is the probabilistic suffix tree of Ron et al. [2].

VOMCs have been used for different applications like classification of transcription factor binding sites, splice sites, and protein families [3,4,5]. Dalevi and

co-workers applied VOMCs to the detection of horizontal gene transfer in bacterial genomes [6]. VOMCs have been used by Bejerano et al. for the segmentation of proteins into functional domains [7], and Slonim and others have shown how to exploit their structure for feature selection [8]. VOMCs do not require a sequence alignment, an advantage over many other sequence models. They dynamically adapt to the data and have been shown to be competitive to HMMs for the task of protein family classification [9]. For the task of DNA binding site classification they outperform commonly used Position Weight Matrices [3,4,10].

There are theoretical results on linear time and space construction of VOMCs [11]. However, currently no linear time implementation has been published and the available tools [6,12] are suboptimal in theory and slow in practice. For example the pst software [12] implements the $\mathcal{O}(\|S\|^2L)$ algorithm due to Ron et al. [2], which is used in the studies [5,9,13]. We believe that three major reasons account for the absence of a linear time implementation: (i) the linear time algorithm proposed by Apostolico and Bejerano appears to be quite complex [11]; (ii) implementations of VOMC construction algorithms have always been explained separately for the methods of Ron et al. [2] and Rissanen [1], which led to the misconception that the first “has the advantage of being computationally more economical than the” second [13]; (iii) it is based on suffix trees which are known to be space-consuming and slow for large sequences due to bad cache locality behaviour.

A lot of research has focused on improving suffix tree construction algorithms by reducing the space requirements [14,15,16,17]. An index called enhanced suffix array can be an efficient replacement for a suffix tree [18]. In this work we will employ some of these strategies and devise and implement improved versions of the Apostolico-Bejerano (AB) algorithm.

In Section 2 we introduce necessary data structures and formulate a general approach to VOMC learning. Both methods [1] and [2] are implemented with the same construction algorithm in Section 3, where the AB algorithm and our improvements are explained. The superiority compared to former algorithms is demonstrated in Section 4.

2 Preliminaries

2.1 Definitions

We consider a collection S of strings over the finite ordered alphabet Σ . Without loss of generality choose $\sigma_i \in \Sigma$, s.t. $\sigma_1 < \dots < \sigma_{|\Sigma|}$. Σ^L is defined as the set of all words of length L over Σ . ϵ is the empty string and $\Sigma^0 = \{\epsilon\}$. The length of a string $s \in S$ is denoted by $|s|$ and $\|S\|$ is defined as the concatenated length of all strings $s \in S$. $|S|$ is the number of strings in the collection, s^R is the reverse of s , and S^R is the collection of the reverse sequences of S . $u[i]$ denotes the i -th character of u . We denote as ur the concatenated string of strings u and r . The *empirical probability* $\tilde{P}(r)$ of a subsequence $r \in S$ is defined as the ratio

$$\tilde{P}(r) = \frac{N_S(r)}{\|S\| - (|r| - 1)|S|} , \quad (1)$$

where $N_S(r)$ is the number of all, possibly overlapping, occurrences of the subsequence r in strings $s \in S$. The denominator is the number of all possible positions of a word of length $|r|$ in S . This defines a probability distribution over words of length l , with $\sum_{r \in \Sigma^l} \tilde{P}(r) = 1$. We define the *conditional empirical probability* $\tilde{P}(\sigma|r)$ of observing the symbol σ right after the subsequence $r \in S$ as

$$\tilde{P}(\sigma|r) = \frac{N_S(r\sigma)}{N_S(r*)}, \quad (2)$$

with $N_S(r*) = \sum_{\sigma \in \Sigma} N_S(r\sigma)$. This can be seen as the relative frequency of σ preceded by r .

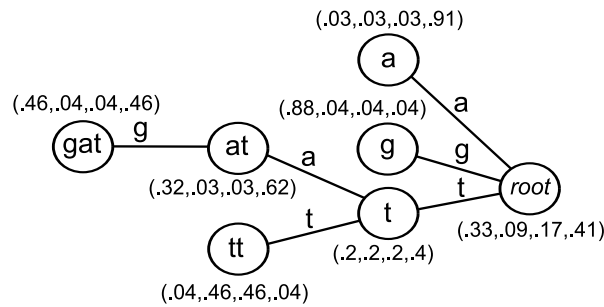


Fig. 1. A PST over the DNA alphabet $\{a,c,g,t\}$. The vector close to each node is the probability distribution for the next symbol, e.g., the symbol t occurs with probability 0.46 after all substrings ending with gat . The prediction of the string $gattaca$ assign a probability to each character: $P(g)P(a)P(t|a)P(t|gat)P(a|tt)P(c|a)P(a) = 0.17 \cdot 0.33 \cdot 0.91 \cdot 0.46 \cdot 0.04 \cdot 0.03 \cdot 0.33$.

2.2 Variable Order Markov Chains

We now introduce the fundamental data structure, needed for learning a VOMC and for prediction of sequences with a VOMC. A convenient representation of a VOMC is the probabilistic suffix tree (PST) [2], sometimes called Context Tree [1]. A PST \mathcal{T} is a rooted tree with edge labels from Σ , and no two child edges have the same character. A node with the concatenated string u , spelled from *node to root*, is denoted \overleftarrow{u} . A node \overleftarrow{u} is called *maximal* if it has less than $|\Sigma|$ children. Each maximal node \overleftarrow{u} in a PST has attached a probability vector $v^u \in [0, 1]^{|\Sigma|}$, with $v_i^u = P(\sigma_i|u)$. Given $r[1]r[2] \dots r[m]$, the next transition is determined as follows: walk down in the PST by following the edges labeled $r[m]r[m-1] \dots$ as long as possible; use the transition probabilities associated to the last node visited.

Example 1. Consider the PST in Fig. 1. The underlying VOMC will map each context ending with character c to $v^c = (0.33, 0.09, 0.17, 0.41)$, the probability

vector of the root. If the context is tat , the transitions correspond to the longest path node $\overleftarrow{\text{at}}$ and are given by $v^{\text{at}} = (0.32, 0.03, 0.03, 0.62)$.

2.3 A General Approach to Construct VOMCs

We divide VOMC learning algorithms [3,4,9,1,2] into two conceptual phases, *Support-Pruning* and *Similarity-Pruning*. In the *Support-Pruning* phase the PST is restricted to representative nodes. In the *Similarity-Pruning* phase redundant nodes of the PST are removed. Tuning the parameters of the *Support-Pruning* and *Similarity-Pruning* phases allows the control of the trade-off between bias and variance for the final VOMC. A strict pruning in each phase shrinks the PST and results in an underfitted VOMC, whereas limited pruning results in a PST that overfits the data. We want to solve the following problem:

Problem 1. Given a string collection S generated by a VOMC, the aim is to find the underlying PST \mathcal{T} using only S .

We will introduce two widely-used solutions to Problem 1 by Rissanen [1] and Ron et al. [2] for learning VOMCs, represented by PSTs, according to our general approach.

Solution 1. Context Tree (S, t, L, K)

Support-Pruning. \mathcal{T} is defined to contain the following nodes:

$$\mathcal{T} \leftarrow \left\{ \overleftarrow{r} \mid r \in \bigcup_{i=0}^L \Sigma^i \text{ and } N_S(r) \geq t \right\} . \quad (3)$$

We denote the first part of the condition in (3) as L -Pruning and the second part as t -Pruning.

Similarity-Pruning. Prune recursively in bottom-up fashion all leaves \overleftarrow{ur} , with $|u| = 1$ and r possibly empty if:

$$\sum_{\sigma \in \Sigma} \left(\tilde{P}(\sigma|ur) \cdot \ln \frac{\tilde{P}(\sigma|ur)}{\tilde{P}(\sigma|r)} \right) \cdot N_S(ur) < K . \quad (4)$$

The original solution proposed by Rissanen [1] set $t = 2$, and it was shown later to be a consistent estimator for any finite t by Bühlmann and Wyner [19].

Example 2. Consider the string set $S = \{\text{gattc, attgata}\}$ and set $t = 2$, $L = 3$, and $K = 0.04$. The *Context Tree* algorithm will build the PST depicted in Fig. 1. All probability vectors of nodes \overleftarrow{r} have been smoothed by adding 0.01 to all $N_S(r\sigma)$, $\sigma \in \Sigma$, before the conditional probabilities are calculated with (2).

Solution 2. Growing PST $(S, P_{\min}, L, \alpha, \gamma_{\min}, k)$

Support-Pruning. Let X be a string collection. Initially \mathcal{T} is empty and $X = \{\epsilon\}$. While $X \neq \emptyset$, repeat the following: (i) pick and remove any $r \in X$; (ii) add \overleftarrow{r} to \mathcal{T} ; (iii) If $|r| < L$, extend X as follows:

$$X \leftarrow X \cup \left\{ \sigma r \mid \sigma \in \Sigma \text{ and } \tilde{P}(\sigma r) \geq P_{\min} \right\} . \quad (5)$$

Similarity-Pruning. Prune recursively in bottom-up fashion all leaves \overleftarrow{ur} , with $|u| = 1$ and r possibly empty if there exists no symbol $\sigma \in \Sigma$ such that

$$\tilde{P}(\sigma|ur) \geq (1 + \alpha)\gamma_{\min} \quad (6)$$

and

$$\frac{\tilde{P}(\sigma|ur)}{\tilde{P}(\sigma|r)} \geq k \quad \text{or} \quad \frac{\tilde{P}(\sigma|ur)}{\tilde{P}(\sigma|r)} \leq \frac{1}{k} . \quad (7)$$

2.4 Suffix Trees

We will use artificial string markers $\$j$ at the end of each string s_j to distinguish the suffixes of strings in a collection $S = \{s_1, \dots, s_{|S|}\}$.

A suffix tree for a collection S over Σ is a rooted tree with edge labels from $(\Sigma \cup \{\$, \dots, \$_{|S|}\})^*$, s.t. every concatenation of symbols from the root to a leaf node yields a suffix of $s_j\$j$ for a string $s_j \in S$. Each internal node has at least two children and no two child edges of a node start with the same character. By this definition, each node can be mapped one-to-one to the concatenation of symbols from the *root to itself*. If that concatenated string is r the node is denoted \overrightarrow{r} . Notice that \overleftarrow{u} is the node with the concatenated string u read from *node to root*. If $u = r^R$, \overrightarrow{r} and \overleftarrow{u} denote the same node. An important concept of suffix trees are *suffix links* [14]. They are used to reach node \overrightarrow{r} from a node $\overrightarrow{\sigma r}$, $\sigma \in \Sigma$. Figure 2.1 shows a suffix tree built for Example 2.

3 Algorithms for VOMC Construction

3.1 The AB Algorithm

The algorithm of Apostolico and Bejerano builds a PST in optimal $O(\|S\|)$ time and space. We will explain the AB algorithm according to our general approach from Section 2.3. The *Support-Pruning* phase is implemented with a suffix tree built on S in linear time and space. Based upon that tree, all contexts which do not fulfill the conditions of the *Support-Pruning* phase are pruned, see Fig. 2.1 and (3),(5). A bottom-up traversal of the suffix tree yields the absolute frequencies and thus the probability vectors for each internal node. However, as the suffix tree is built over S and not S^R , going down in the suffix tree means to extend a subsequence to the right, whereas context extensions are subsequence extensions to the left. In other words, the father-son relation of the PST is unknown, an essential information for the *Similarity-Pruning* phase. Hence, so-called reverse suffix links (rsufs) are introduced, which are the opposite of suffix links, see Fig. 2.3. Via rsufs all existent nodes $\overrightarrow{\sigma r}$, $\sigma \in \Sigma$, can be reached from each node \overrightarrow{r} . Walking the reverse suffix link from \overrightarrow{r} to $\overrightarrow{\sigma r}$ is equivalent to walking from father \overleftarrow{r} to son $\overleftarrow{\sigma r}$ in the PST.

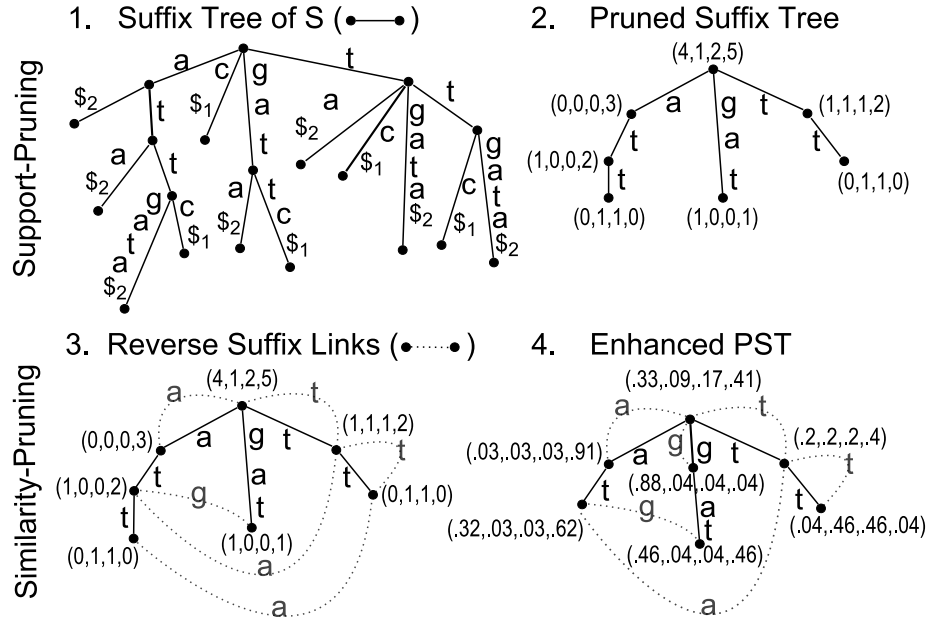


Fig. 2. Overview of our implementation of the AB algorithm [11] for PST construction. It can be divided into two pruning phases: Support-Pruning and Similarity-Pruning. The figure shows the detailed construction for the parameters given in Example 2. A detailed description can be found in Example 3.

Reverse suffix links, just as suffix links, can be constructed for a suffix tree in linear time [20]. Using rsufs and after adding missing nodes,⁴ it can be shown that the *Similarity-Pruning* phase takes only $\mathcal{O}(\|S\|)$ time.

Example 3. Figure 2 shows how the AB algorithm constructs the PST for the parameters given in Example 2. In Fig. 2.1 the complete suffix tree of S is constructed. In step 2, a bottom-up traversal determines for all nodes the absolute frequencies and nodes \vec{r} , with $N_S(r) < 2$, are pruned. Reverse suffix links are added in Fig. 2.3, the start of the *Similarity-Pruning* phase. The node \vec{att} is pruned, as (4) is satisfied, and the missing node \vec{g} is added in step 4. Finally all vector entries are smoothed, see Example 2. The same PST as in Fig. 1 is obtained when only the dotted lines, the rsufs, are considered.

We call the resulting automaton *enhanced PST*, as it allows a traversal from node \vec{r} to nodes $\vec{\sigma r}$ via rsufs and to nodes $\vec{r\sigma}$ via suffix tree edges, for all $\sigma \in \Sigma$. The enhanced PST can be modified to predict a new sequence of length m in

⁴ It can happen that new auxiliary nodes need to be added to the suffix tree. Branching nodes in a PST are not necessarily branching in the suffix tree. However, the missing nodes, each a target of a reverse suffix link, can be created in $\mathcal{O}(\|S\|)$ time [11].

$\mathcal{O}(m)$ time instead of $\mathcal{O}(mL)$ for a PST [11], which is another advantage of the algorithm. We have implemented the AB algorithm using the *eager* write-only top-down (WOTD)-algorithm for suffix tree construction, one of the fastest and most space efficient suffix tree construction algorithms [15].

3.2 Adaptive Algorithms for VOMCs

VOMCs select the abundant contexts in the data, the carriers of information. They account for a small subset of suffix tree nodes, the pruned suffix tree, which is determined by a traversal of the suffix tree. In the following we suggest two different approaches to attain the pruned suffix tree more efficiently than constructing the complete suffix tree. $N_S(r)$ and the absolute frequencies can be obtained in a top-down traversal. After attaining the pruned suffix tree, we proceed with steps 3 and 4 of the AB algorithm; add reverse suffix links and prune similar nodes. The second approach is depicted for Solution 1 in Algorithm 1.

An Adaptive VOMC Algorithm with Enhanced Suffix Arrays. The suffix array stores the starting positions of all lexicographically ordered suffixes of a string [16]. We use the deep shallow algorithm by Manzini and Ferragina for suffix array construction [21]. An enhanced suffix array (ESA) is a suffix array which has two additional arrays, the lcp-table and the child-table which are used to simulate a top-down traversal on a suffix tree of S in linear time and space [18]. The ESA is an efficient replacement for the suffix tree in the first step of the *Support-Pruning* phase, see Fig. 2.1. It can be used to traverse a suffix tree top-down using $9 \cdot \|S\|$ bytes of memory [18], assuming that L is always smaller than 255. The traversed nodes are added to a new graph, the pruned suffix tree. This tree is used in the next steps, depicted in Fig. 2.2–2.4. After the traversal, the ESA is discarded. We refer to this algorithm as **AV-1**.

An Adaptive VOMC Algorithm with Lazy Suffix Trees. An approach more appealing than the previous one is to avoid building the complete ESA, but build only the parts of the suffix tree to be traversed. That means to entirely skip the first step of the *Support-Pruning* phase. The lazy WOTD-algorithm is perfectly suited to restrict the buildup of the suffix tree as it expands nodes top-down. A lazy suffix tree is a suffix tree whose nodes are created on demand, i.e. when they are visited the first time. Giegerich et al. introduced a lazy suffix tree data structure [22] that utilizes the WOTD-algorithm [15,22] for the on-demand node creation. In the beginning, T contains only the root node and is iteratively extended by suffix tree nodes, to at most the entire suffix tree.

After creating a node \vec{r} in T the values $|r|$ and $N_S(r)$, relevant for the *Support-Pruning* phase, are known. Thus the restriction for length L and $N_S(r)$ can be included to constrain the top-down construction, see lines 1–9 of Algorithm 1. This does not only save construction time, but also reduces the amount of space needed. Steps 3 and 4 of the AB algorithm are realized in lines 10–13. The adaptive algorithm with lazy suffix trees is referred to as **AV-2**.

Algorithm 1: createEnhancedPST(S, t, L, K)

Input : string set S over Σ , min. support t , max. length L , pruning value K
Output : enhanced PST \mathcal{T} for Solution 1
// Support-Pruning
1 Let X be a string collection and \mathcal{T} be a suffix tree containing only the root node.
2 $X \leftarrow \{\epsilon\}$
3 **foreach** $x \in X$ **do**
4 $X \leftarrow X \setminus \{x\}$
5 **if** $|x| < L$ **then**
6 **foreach** $\sigma \in \Sigma$ with $N_S(x\sigma) \geq t$ **do**
7 Let $y \in \Sigma^*$ be the longest string, s.t. $N_S(x\sigma y) = N_S(x\sigma)$ and $|x\sigma y| \leq L$
8 Insert $\overrightarrow{x\sigma y}$ into \mathcal{T} .
9 $X \leftarrow X \cup \{x\sigma y\}$
// Similarity-Pruning
10 Insert auxiliary nodes into \mathcal{T} [11] and add probability vectors to all nodes.
11 **foreach** $\overrightarrow{x}, \overrightarrow{\sigma x} \in \mathcal{T}$, $\sigma \in \Sigma$ **do**
12 Add a reverse suffix link from \overrightarrow{x} to $\overrightarrow{\sigma x}$.
13 In a bottom-up traversal using only the reverse suffix links of \mathcal{T} remove all nodes satisfying (4).
14 **return** \mathcal{T}

4 Results

We compared the runtime of our algorithms within two experiments. In the first we investigated the improvement of the AV-1 and AV-2 algorithm compared to the AB algorithm. In the second, we compared our new tool with two existing software tools. For the *Growing PST* solution we used the **RST** algorithm [12] and for the *Context Tree* solution the **Dal** algorithm [6]. The algorithms were applied to data from applications mentioned in the introduction: three protein families from the classification experiment of Bejerano et al. [9], two bacterial genomes as in [6] retrieved from GenBank [23], a set of human promoter sequences [24], and the complete human proteome from Uniprot [25], see Table 2. Experiments were conducted under Linux on an Intel Xeon 3.2 GHz with 3 GB of RAM. Output of all programs was suppressed. For all tests of the *Context Tree* algorithm we fixed $K = 1.2$, and for the *Growing PST* algorithm we fixed $\alpha = 0$, $\gamma_{\min} = 0.01$, $k = 1.2$.

First we compared the runtime of the different *Support-Pruning* phases of our implemented algorithms AB, AV-1, and AV-2. The results are shown for four data sets with varying L , and three values for P_{\min} or t in Fig. 3. The AB algorithm performs worse than the two adaptive algorithms for all sequences. The AV-2 algorithm outperforms the AB and AV-1 algorithms for restrictive parameters, but for small values of t the AV-2 implementation of the *Context Tree* algorithm becomes less efficient and is outperformed by the AV-1 algorithm. For the values of t in the H.inf. plot in Fig. 3 the runtime of the AV-2 algorithm

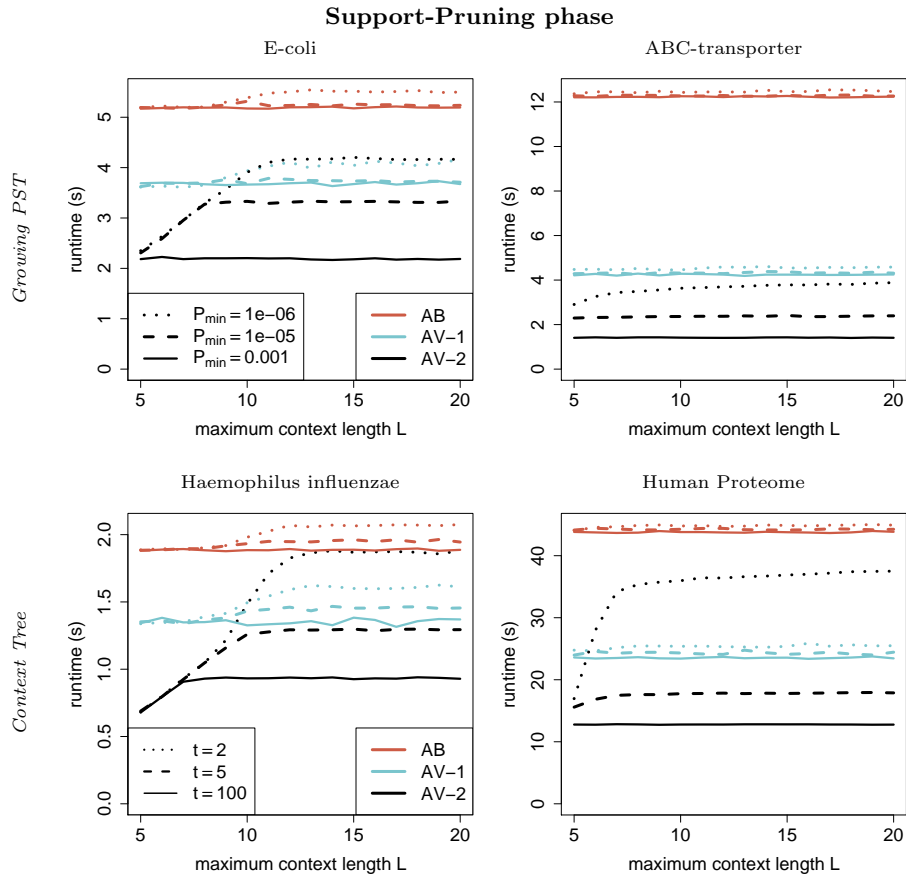


Fig. 3. Runtime results for different *Support-Pruning* phases of our implemented algorithms AB, AV-1, and AV-2, averaged over three runs.

depends on the L -Pruning up to context length 7, whereas for long contexts the t -Pruning dominates the L -Pruning. On the proteome data set a similar behaviour is already visible for smaller values of L , because of the larger alphabet size.

Second we compared the runtime for the complete VOMC construction of our implementations with the Dal algorithm and the RST algorithm. Table 1 reveals the superiority of our approach. The Dal algorithm is up to 11600 times slower than the AV algorithms and crashes for some instances due to insufficient memory. See Table 1 for the memory consumption for one such instance. The RST algorithm is up to 2700 times slower than our algorithms not including the parameter settings, where the RST algorithm was terminated after 100 minutes. However, our algorithms are efficient for all parameter settings and alphabets.

Table 1. Runtimes in seconds of our implementations for *Context Tree* and *Growing PST* solutions on various data sets and parameter settings.

<i>Runtime Context Tree Algorithms</i>												
(L, t)	(5,2)				(9,5)				(15,100)			
Name	Dal	AB	AV-1	AV-2	Dal	AB	AV-1	AV-2	Dal	AB	AV-1	AV-2
H.inf.	6.56	2.04	1.72	0.74	28.9	3.98	3.78	3.12	485	2.16	1.75	1.12
E.coli	15.6	5.58	4.60	2.54	50.6	8.46	7.85	6.89	†	5.91	5.00	3.75
Promoters	76.5	50.4	31.1	19.3	179	53.9	34.9	36.1	†	52.3	32.8	34.4
7tm	83.1	1.62	1.61	1.49	339	0.82	0.71	0.71	812	0.21	0.14	0.07
ig	111	3.82	3.33	3.35	489	2.41	1.93	1.89	1200	0.98	0.46	0.20
ABC	618	41.2	31.3	30.8	†	30.3	20.5	18.5	†	16.2	5.97	2.75
Proteome	1810	59.2	51.0	36.8	†	60.3	51.6	39.7	†	39.1	27.4	14.8

<i>Runtime Growing PST Algorithms</i>												
(L, P_{\min})	$(5, 10^{-6})$				$(9, 10^{-5})$				(15, 0.001)			
Name	RST	AB	AV-1	AV-2	RST	AB	AV-1	AV-2	RST	AB	AV-1	AV-2
H.inf.	119	2.07	1.60	0.71	–	2.71	2.13	1.68	102	2.00	1.53	0.70
E.coli	305	5.42	4.29	2.41	–	6.08	5.03	4.10	252	5.46	4.19	2.26
Promoters	1240	48.7	29.3	18.0	–	49.4	29.7	29.1	1020	48.8	28.4	15.4
7tm	1710	4.40	4.39	4.34	2400	2.42	2.37	2.36	4.20	0.21	0.13	0.06
ig	3260	7.77	7.21	7.28	4900	2.51	1.91	1.81	11.3	0.93	0.39	0.12
ABC	–	22.2	11.8	9.47	–	16.7	6.17	3.43	85.5	15.9	5.20	1.52
Proteome	–	43.6	31.6	20.2	–	37.7	25.0	12.7	392	36.7	24.0	7.39

† Program terminated due to insufficient internal memory (> 3 GB).

– Program run for more than 6000 s.

5 Discussion and Conclusions

We achieved an order of magnitude improvement for VOMC construction algorithms compared to two previous algorithms. In addition, we further improved the overall time and space consumption of [11] by replacing the suffix tree with more efficient and problem-oriented index data structures. All algorithms are implemented with SeqAn [26] in the *Pisa* tool, which is publicly available at <http://www.seqan.de/projects/pisa.html>. To our knowledge it is the most efficient tool for VOMC learning in the bioinformatics community.

In our experiments the lazy suffix tree approach [15] is faster than the ESA approach [18] and consumes roughly half of the memory. Improving the construction time of the suffix array, one of three tables of the ESA, is a topic of current research and an expanding field, outside the scope of this paper.

Other possible improvements of the *Support-Pruning* phase like a q-gram index or length limited suffix tree construction in linear time, e.g. [27], might be considered, but from our experience the lazy suffix tree is the most robust data structure for our general implementation, as it works for large alphabets and deep contexts as well.

Table 2. Data sets used in the experiments and memory usage of the *Context Tree* algorithms.

<i>Memory Usage in MB (L = 15, t = 100)</i>							
Name	Description	$\ S\ $ (mb)	$ \Sigma $	Dal	AB	AV-1	AV-2
H.inf.	Haemophilus influenzae	1.9	5	2010	59.9	65.4	31.6
E.coli	Escherichia coli	4.6	5	†	129	159	76.2
Promoters	Human promoters	22.5	5	†	633	550	373
7tm	transmembrane family	0.2	24	1160	6.57	7.38	3.93
ig	Immunoglobulin family	0.5	24	1540	16.4	18.7	11.8
ABC	ABC transporter family	4.3	24	†	118	156	81.1
Proteome	Human proteome	17.3	24	†	456	503	306

In the future we will investigate extensions to the approach presented here to build more advanced types of VOMCs, which can learn also subset relations e.g. the approach presented by Leonardi [13].

References

1. Rissanen, J.: A universal data compression system. *IEEE Transactions on Information Theory* **29** (1983) 656 – 664
2. Ron, D., Singer, Y., Tishby, N.: The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning* **25** (1996) 117–149
3. Ben-Gal, I., Shani, A., Gohr, A., Grau, J., Arviv, S., Shmilovici, A., Posch, S., Grosse, I.: Identification of transcription factor binding sites with variable-order Bayesian networks. *Bioinformatics* **21**(11) (Jun 2005) 2657–2666
4. Zhao, X., Huang, H., Speed, T.P.: Finding short DNA motifs using permuted Markov models. *J. Comput. Biol.* **12**(6) (2005) 894–906
5. Ogul, H., Mumcuoglu, E.U.: SVM-based detection of distant protein structural relationships using pairwise probabilistic suffix trees. *Comput. Biol. Chem.* **30**(4) (Aug 2006) 292–299
6. Dalevi, D., Dubhashi, D., Hermansson, M.: Bayesian classifiers for detecting HGT using fixed and variable order markov models of genomic signatures. *Bioinformatics* **22**(5) (Mar 2006) 517–522
7. Bejerano, G., Seldin, Y., Margalit, H., Tishby, N.: Markovian domain fingerprinting: statistical segmentation of protein sequences. *Bioinformatics* **17**(10) (Oct 2001) 927–934
8. Slonim, N., Bejerano, G., Fine, S., Tishby, N.: Discriminative feature selection via multiclass variable memory Markov model. *EURASIP J. Appl. Signal Process.* **2003**(1) (2003) 93–102
9. Bejerano, G., Yona, G.: Variations on probabilistic suffix trees: statistical modeling and prediction of protein families. *Bioinformatics* **17**(1) (Jan 2001) 23–43
10. Posch, S., Grau, J., Gohr, A., Ben-Gal, I., Kel, A.E., Grosse, I.: Recognition of cis-regulatory elements with vomat. *J. Bioinform. Comput. Biol.* **5**(2B) (Apr 2007) 561–577
11. Apostolico, A., Bejerano, G.: Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. *J. Comput. Biol.* **7**(3-4) (2000) 381–393

12. Bejerano, G.: Algorithms for variable length Markov chain modeling. *Bioinformatics* **20**(5) (Mar 2004) 788–9
13. Leonardi, F.G.: A generalization of the PST algorithm: modeling the sparse nature of protein sequences. *Bioinformatics* **22**(11) (Jun 2006) 1302–7
14. Kurtz, S.: Reducing the space requirement of suffix trees. *Software Pract. Exper.* **29**(13) (1999) 1149–1171
15. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. *Software Pract. Exper.* **33**(11) (2003) 1035–1049
16. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.* **22**(5) (1993) 935–948
17. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms* **3**(2) (2007) 20
18. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms* **2** (2004) 53–86
19. Bühlmann, P., Wyner, A.J.: Variable length Markov chains. *Ann. Statist.* **27**(2) (1999) 480–513
20. Maaß, M.G.: Computing suffix links for suffix trees and arrays. *Inf. Process. Lett.* **101**(6) (2007) 250–254
21. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. *Algorithmica* **40**(1) (2004) 33–50
22. Giegerich, R., Kurtz, S.: A comparison of imperative and purely functional suffix tree constructions. *Sci. Comput. Program.* **25** (1995) 187–218
23. Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Wheeler, D.L.: GenBank. *Nucleic Acids Res* **36**(Database issue) (Jan 2008) D25–D30
24. Fitzgerald, P.C., Sturgill, D., Shyakhtenko, A., Oliver, B., Vinson, C.: Comparative genomics of drosophila and human core promoters. *Genome Biol.* **7** (2006) R53
25. The UniProt Consortium: The Universal Protein Resource (UniProt). *Nucl. Acids Res.* **36**(suppl.1) (2008) D190–195
26. Döring, A., Weese, D., Rausch, T., Reinert, K.: SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics* **9** (2008) 11
27. Schulz, M.H., Bauer, S., Robinson, P.N.: The generalised k-Truncated Suffix Tree for time- and space- efficient searches in multiple DNA or protein sequences. *Int. J. Bioinform. Res. Appl.* **4**(1) (2008) 81–95