INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

On the Representative Power of Commented Markov Models

Reinhard Blasig and Gerald Fahner
blasig/fahner@icsi.berkeley.edu

TR-96-034

August 1996

Abstract

A CMM (Commented Markov Model) is a learning algorithm to model and extrapolate discrete sequences. The learning involves the inferences of *objects*, *variables* and *classes*, describing the sequences. In this paper, all sequences considered will be character sequences. As pointed out in an earlier paper [2], the structures utilized by CMM are powerful enough to represent and evaluate any *primitive recursive function*. This paper will provide a formal proof of this claim. We will therefore concentrate on the issues of representation and leave the issues of CMM induction aside.

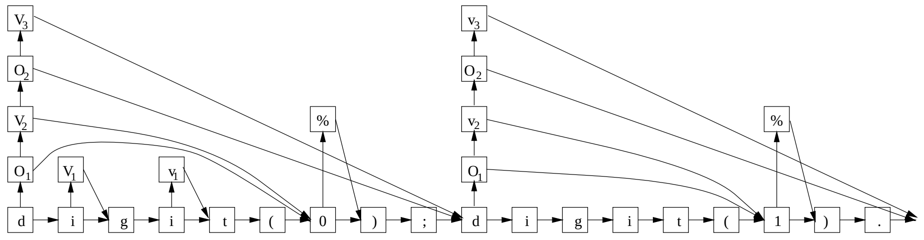
1 Introduction

To establish the above mentioned proof, this paper will proceed in three steps. First, the relevant concepts of CMM are presented (for a motivation and a more extended introduction of these concepts see [2]). Then we review the definition of "primitive recursive functions" (*prf*s) and show how these are represented by CMM character strings. Finally, some theorems are proven, that establish the equivalence of CMM extrapolating a string representation of a *prf* and of evaluating a *prf*.

2 CMM concepts

The input processed by CMM is a discrete sequence $(s_1, s_2, \ldots)$. Here we will consider the case where the sequence consists of characters. To represent the input, CMM generates a directed acyclic graph called *sequence graph*. Figure 1 shows an exemplary graph created for the character string

$$(s_1, s_2, \ldots) = \text{‘}\texttt{digit(0);digit(1).}\text{’} \tag{1}$$



2.1 Objects, Variables and Classes

The nodes in the graph are the *sequence cells*, each of which represents one of the following items (explained along with the matching relation between sequence cells):

- an object, which can be either a *primitive object* (i.e. a single letter, digit or any other character) or a *composed object* denoted by O_i, which in turn represents a sequence of shorter items. A main part of the learning capabilities of CMM results from its ability to construct complex objects from a sequence of simpler items. Two sequence cells representing the same object can be *matched*.

- a variable: Variables come in pairs (V_j, v_j), where the V_j marks the first position of a reoccuring sequence and the v_j mark the succeeding ones. This sequence may consist of a single object, e.g. the character `'i'` for the (V_1, v_1)-pair in the example graph. The matching relation for variables is defined so that a pair of sequence cells representing a V_i and a corresponding v_i can be matched to two sequence cells representing a V_j/v_j-pair.

- a class: Originally, classes just collect objects or object sequences, that can be found in a similar context, but that cannot be matched themselves. For example, the cells with the label C_1 make up a class, that consists of the to elements '`0`' and '`1`', which have both been found in the context '`digit(...)`'. A sequence cell representing a class C_k can be matched to any object known to be a member of C_k.

In this paper, we will not deal with the inference mechanisms used to create the sequence graph, to infer objects, variables or classes. As far as classes are concerned, we will simply consider them as set of objects, where the set can be described either by enumerating the corresponding objects or by any other suitable way. Again, for more details see [2].

2.2 Sequence Prediction with CMM

The fundamental operation of CMM is the search for matches of the current input sequence with sequences already known to CMM. This involves two separate matching tasks. First, the input will be matched with sequences corresponding to the objects already known to CMM. These sequences are stored in a dictionary. Secondly, the input sequence will be matched to itself in the search for reoccurring subsequences in the input. This latter matching task is necessary to infer new objects, variables and classes to model the input sequence. As we already pointed out, this paper will disregard the issues of learning, so we will consider a CMM with only the matching of the input sequence with the dictionary objects being operative.
In this setting, sequence prediction with CMM amounts to continuously searching for the dictionary object, that matches the recent input sequence best. For an object to match the input sequence the last characters of the input must be matched with the beginning of the object. The more characters of the input sequence match, i.e. the more characters of the input sequence are explained by the presence of the matching object, the better the match is. The object providing the best match will then be used to make a prediction about the coming input characters. The search for the best match and thus the prediction is preformed after each new character being added to the input sequence. The new character may be either provided by an external source or by CMM's prediction.

3 Primitive Recursive Functions

Primitive recursive functions are recursively defined according to the following five points:

1. The number 0 is a *prf*.

2. If x is primitive recursive, then the successor $s(x)$ of x is a *prf*. The successor function can be used to represent any natural number: $1 = s(0)$, $2 = s(s(0))$, …, $i = s^i(0)$.
3. If the expressions $x_1, \ldots, x_n$ are primitive recursive, then the projection function $p_i^n(x_1, \ldots, x_n) = x_i$ is a *prf*.
4. The composition $f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ of *prf*s $g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)$ is a *prf*.
5. primitive recursion: For *prf*s g and h the following function f is also a *prf*:

$$f(x_1, \ldots, x_n) = \begin{cases} g(x_1, \ldots, x_{n-1}) & \text{for } x_n = 0 \\ h(x_1, \ldots, x_n, f(x_1, \ldots, x_{n-1}, x_n - 1)) & \text{for } x_n \neq 0. \end{cases}$$

The following example definition of a function $add(x_1, x_2)$ shows that the summation of two natural numbers is primitive recursive:

$$add(x_1, x_2) = \begin{cases} p_1^1(x_1) & \text{for } x_n = 0 \\ s(p_3^3(x_1, x_2, add(x_1, x_2 - 1))) & \text{for } x_n \neq 0. \end{cases}$$

Since $p_1^1()$ and $s(p_3^3())$ are both primitive recursive, so is $add()$.

3.1 Evaluation of Primitive Recursive Functions

From the above definition it follows that any *prf* either has the form $s^i(0)$ with $i \geq 0$ or it contains a subexpression e, which has either the form $p_i^n(x_1, \ldots, x_n)$ (projection), or $f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ (composition), or $f(x_1, \ldots, x_n)$ with a primitive recursive definition for f. In the case of a projection or primitive recursion, we say that an evaluation step is performed if the subexpression – which has the form of the lhs of (3) or (5) – is substituted by the rhs of (3) or (5), respectively. In the case of function composition, the evaluation step either involves evaluating one of the g_i (i.e. the arguments of the composed function f) or by evaluating f itself[1]. This can be done, since f must be defined in terms of the successor function, projection and/or primitive recursion. Since *prf*s are total functions, the evaluation will terminate after a finite number of steps and the result will be of the form $s^i(0)$.
The evaluation of a *prf* is performed as a sequence of evaluation steps. As an example consider the evaluation of the expression $add(s(s(s(0))), s(s(0)))$, with the function $add()$ as defined above:

$$\begin{array}{l} add(s(s(s(0))), s(0)) \\ s(p_3^3(s(s(s(0))), s(0), add(s(s(s(0))), 0))) \end{array}$$

[1]We say that function evaluation is *strict*, if f is evaluated only after the evaluation of its arguments is completed, i.e. all arguments have the form $s^i(0)$.

$$s(p_3^3(s(s(s(0))), s(0), p_1^1(s(s(s(0))))))$$
$$s(p_3^3(s(s(s(0))), s(0), s(s(s(0)))))$$
$$s(s(s(s(0))))$$

Note, that the evaluation sequence is not necessarily unique, even if we demand the evaluation to be strict (as is the above example). However, the result is unique and has the form $s^i(0)$, as already mentioned before.

3.2 String representations of *prf*s

Primitive recursive expressions can be perceived as having a tree structure. The leaves of the tree are all labelled '0', and the internal nodes represent functions, with their descendents stating the function arguments. Of course, primitive recursive expressions can also be represented as plain linear character sequences, e.g. '`add(s(s(s(0))),s(0))`'. In the following we will use the simpler representation '`ss...s0`' instead of '`s(s(...s(0))...)`', so the string representation of the expression $add(s(s(s(0))), s(0))$ would be '`add(sss0,s0)`'. Here is the definition of the *string representation* of a primitive recursive expression E:

Definition: The string representation $\mathcal{S}(E)$ of a primitive recursive expression E is recursively defined as follows:

- If $E = 0$, then $\mathcal{S}(E) =$'`0`'.
- If $E = s(E_1)$, then $\mathcal{S}(E) =$'`s`$\mathcal{S}(E)$'.
- If $E = f(E_1, \ldots, E_n)$ and f is not the successor function s, then $\mathcal{S}(E) =$'$\mathcal{F}(f)$`(`$\mathcal{S}(E_1)$`,... ,`$\mathcal{S}(E_n)$`)`'.
  $\mathcal{F}(f)$ is a sequence of characters to represent the function name of f, e.g. $\mathcal{F}(add)$ = '`add`'. A special case are the projection functions P_n^i. For these we set $\mathcal{F}(P_n^i)$ = '`P`$III|NNN$', where III and NNN are sequences of digits, that represent i and n, e.g. $\mathcal{S}(P_{12}^4(\ldots) =$`P4|12(...)`'.

  The following restriction applies to the selection of function names: The set $\mathcal{N}$ of all function names has to be suffix free, i.e. there must be no two functions $f \neq g$ with the string $\mathcal{F}(f)$ being a suffix of $\mathcal{F}(g)$.[2]

When CMM is to evaluate a primitive recursive expression, it is restricted to dealing with this character string. Again, the evaluation process with consist of a number of evaluation steps, generating a sequence of primitive recursive expressions. To separate successive expressions, we enclose them by square brackets. So an expression to be evaluated migth look like this:

$$\text{`}[\texttt{add(sss0, s0)}]' \tag{2}$$

[2]Note, that we do not include the set $\mathbf{s}^i$, $i > 0$ of successor function representations in $\mathcal{N}$.

The evaluation sequence would then be: '`[add(sss0,s0)][sP3|3(sss0,s0,add(sss0,0))][sP3|3(sss0,s0,P1|1(sss0))]` `[sP3|3(sss0,s0,sss0)][ssss0]`'
The important point is that CMM, given the string representation of a primitive recursive expression as input, will extrapolate the corresponding input sequence by generating new expressions and thus performing one evaluation step after the other. Now it needs to be shown that CMM's matching process together with the concepts of objects, variables and classes is powerful enough to perform this extrapolation.

3.3 Function Evaluation as String Extrapolation

Let's assume that CMM knows the following objects:

- $\texttt{[}V_\alpha\texttt{P}III|NNN\texttt{(\%,}\ldots\texttt{,\%,}V_p\texttt{,\%,}\ldots\texttt{,\%)}V_\omega\texttt{][}v_\alpha v_p v_\omega\texttt{]}$, where III and NNN are used as in section (3.2).
- $\texttt{[}V_\alpha\texttt{F(}V_1\texttt{,}\ldots\texttt{,}V_{n-1}\texttt{,0)}V_\omega\texttt{][}v_\alpha\texttt{G(}v_1\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{)}v_\omega\texttt{]}$
- $\texttt{[}V_\alpha\texttt{F(}V_1\texttt{,}\ldots\texttt{,}V_{n-1}\texttt{,s}V_n\texttt{)}V_\omega\texttt{][}v_\alpha\texttt{H(}v_1\texttt{,}\ldots\texttt{,}v_{n-1}\texttt{,s}v_n\texttt{,F(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)}v_\omega\texttt{]}$

In the following, these objects will be referred to as the *dictionary objects.* Again, we do not consider the question, where they come from. We just assume that there has been some (induction) algorithm that put them into CMM's dictionary.
CMM matching restriction: the CMM matching algorithm has to be restricted in that V_α and V_ω may match any character sequence that does not contain a '`[`' or a '`]`'. Additionally, all other variables (i.e. V_p and the V_i) and the placeholders '`%`' may only be matched to strings of the form '$\texttt{s}^i\texttt{0}$' with $i \geq 0$. These restrictions are expressible within the CMM class concept, that can be used to describe the possible values of variables and placeholders.
Before we start with the proof, we want to provide an intuition on how CMM actually performs the evaluation. First of all it must be noted that the first of the above three objects represents a whole object family. These objects are supposed to implement all the projection functions $p_i^n()$, $n > 0$ and $0 \leq i \leq n$, that are needed in the course of the evaluation process. The second and third of the above objects will be used to evaluate primitive recursion. Here the symbols '`F`', '`G`' and '`H`' represent arbitrary functions names.
Function composition is handled by the introduction of the variable pairs V_α/v_α and V_ω/v_ω. Let's reconsider the above evaluation sequence for the string '`[add(sss0,s0)]`': Of course, CMM has to know how the function `add()` is defined. According to the scheme presented above the definition is provided by the following objects:

- $O_1 = \texttt{[}V_\alpha\texttt{add(}V_1\texttt{,0)}V_\omega\texttt{][}v_\alpha\texttt{P}^1_1\texttt{(}v_1\texttt{)}v_\omega\texttt{]}$

- $O_2 = \texttt{[}V_\alpha\texttt{add(}V_1\texttt{,s}V_2\texttt{)}V_\omega\texttt{]}\ \texttt{[}v_\alpha\texttt{sP}^3_3\texttt{(}v_1\texttt{,s}v_2\texttt{,add(}v_1\texttt{,}v_2\texttt{))}v_\omega\texttt{]}$
  Here, only two different projection functions are utilized, which can be readily defined by
- $O_3 = \texttt{[}V_\alpha P^1_1\texttt{(}V_p\texttt{)}V_\omega\texttt{]}\ \texttt{[}v_\alpha v_p v_\omega\texttt{]}$
- $O_4 = \texttt{[}V_\alpha P^3_3\texttt{(\%,\%,}V_p\texttt{)}V_\omega\texttt{]}\ \texttt{[}v_\alpha v_p v_\omega\texttt{]}$

Now, when CMM is given the input sequence '`[add(sss0,s0)]`', it tries to find a match with the objects it already knows. Remember, that since we want to extrapolate the input string after the character '`]`', we are only interested in matches that include the final part of this string, i.e.

$$\begin{array}{r}
\texttt{]}\\
\texttt{)]}\\
\texttt{0)]}\\
\texttt{s0)]}\\
\texttt{,s0)]}\\
\texttt{0,s0)]}\\
\texttt{s0,s0)]}\\
\texttt{ss0,s0)]}\\
\texttt{sss0,s0)]}\\
\texttt{(sss0,s0)]}\\
\texttt{d(sss0,s0)]}\\
\texttt{dd(sss0,s0)]}\\
\texttt{add(sss0,s0)]}\\
\texttt{[add(sss0,s0)]}
\end{array}$$

Also, an object will only be extrapolated by CMM, if this object has been matched from its beginning. Since all the above defined objects start with the character '`[`', the match will necessarily start with a '`[`' in the input sequence. In consequence, the extrapolation will be based on a match of the whole input sequence '`[add(sss0,s0)]`', and the only way to match this to one of the above objects is by chosing O_2, instantiating V_α and V_ω with the empty string, V_1 with '`sss0`' and V_2 with '`0`'. The input sequence can now be extrapolated by appending the rest of O_2, which results in

$$\text{`}\texttt{[add(sss0,s0)][sP3|3(sss0,s0,add(sss0,0))]}\text{'}. \tag{3}$$

The procedure of finding a match continues, as the extrapolation produces one character after the other. During this process O_2 keeps the quality of being the best match to the input sequence, until the end of O_2 is reached by producing the closing square bracket. As before, the new best match has to start with an opening square bracket in the input sequence. But as V_α and V_ω must not be instantiated with a string containing a '`[`' or a '`]`', there is no way to match the extended input string to

one of the objects $O_1, \ldots, O_4$.[3]
After producing the final '`]`' the best match thus becomes the match of '`[sP`$^3_3$`(sss0,s0,add(sss0,0))]`' with the first part '`[`V_α`add(`V_1`,0)`V_ω`]`' of object O_1, where V_α ='`sP`$^3_3$`(`sss`0,s0,`', V_ω represents the empty string and V_1 ='`sss0`'. The input string can now be extrapolated according to O_1, which yields:

$$\text{`}\mathtt{[add(sss0,s0)][sP^3_3(sss0,s0,add(sss0,0))][sP^3_3(sss0,s0,P^1_1(sss0))]}' \qquad (4)$$

The two final steps of the calculation are performed by matching '`[sP`$^3_3$`(sss0,s0,P`$^1_1$`(sss0))]`' with '`[`$V_\alpha P^1_1$`(`V_p`)`V_ω`]`' (the first portion of O_3 producing the sequence '`[sP`$^3_3$`(sss0,s0,sss0)]`', which is in turn matched to '`[`$V_\alpha P^3_3$`(%,%,`V_p`)`V_ω`]`' of O_4. The last created expression and the result is '`ssss0`'.

4 The Representative Power of CMM

The objective of this section is to prove that CMM has the representative power of primitive recursive functions. We will do this by showing that *prf* evaluation is equivalent to CMM string extrapolation, when applying the dictionary objects as defined in section 3.3. To make this more concrete, we will show that iff E is a primitive recursive expression which can be valuated to[4] $s^i(0) = s(s(\ldots(0)\ldots)$ with $i \geq 0$, and iff S is the string expression representing E, then there is a CMM (given by objects $O_1, \ldots, O_n$) which will extrapolate S to produce the sequence '`s`^{i}`0`'='`ss...s0`'.
The proof will consist of two parts, represented by the following two theorems:
Theorem 1: Given a primitive recursive expression E and objects as defined in section (3.3). If there is a strict evaluation step evaluating E to E^e, then there is an object O_i with its first part matching '`[`$\mathcal{S}(E)$`]`' and the second (and last) part matching '`[`$\mathcal{S}(E^e)$`]`'.
In other words, the object O_i enables CMM to extrapolate '`[`$\mathcal{S}(E)$`]`' by producing '`[`$\mathcal{S}(E^e)$`]`'.
Theorem 2: If the first part of an object O_j represents a match for '`[`$\mathcal{S}(E)$`]`' with the primitive recursive expression E, then the second part of O_j matches '`[`$\mathcal{S}(E^e)$`]`', where E^e can be derived from E by a strict evaluation step.
Corollary: Since the evaluation sequence of any primitive recursive expression is finite, the two theorems imply that the character sequence produced by CMM will also terminate.
Proof of theorem 1: The existence of a strict evaluation step turning E into E^e implies that there is a subexpression $E_s = f(s^{j_1}(0), \ldots, s^{j_n}(0))$ in E, that has been transformed into the corresonding subexpression E^e_s of E^e. In the tree representation of E, this evaluation step would be performed by substituting the subtree representing E_s with E^e_s yielding the tree representation of E^e.

[3]Except of course matching the whole input to the whole object O_2. This match is discarded, because it does not provide an extrapolation for the input.

[4]Since primitive recursive functions are total, this value exists and is unique.

Let's look at f more closely. The function f is a composition function, which – by the definition of primitive recursive functions – must be defined as either the successor function, a projection function or a primitive recursion. In this special case, f cannot be the successor function, because in that case E_s would have the form $s^j(0)$, which cannot be further evaluated.

If f is a projection function, then $E_s = P_n^i(s^{j_1}(0),\ldots,s^{j_n}(0))$ and $E_s^e = s^{j_i}(0)$. By the definition of string representations, the string $\mathcal{S}(E)$ has the substring $\mathcal{S}(E_s)$ = '$\mathtt{P}III|NNN(\mathtt{s}^{j_1},\ldots,\mathtt{s}^{j_n})$' and $\mathcal{S}(E^e)$ has the substring $\mathcal{S}(E_s^e)$ ='$\mathtt{s}^{j_i}$'. Since the evaluation step only changes the subexpressions of the corresponding substrings, respectively, the sequence '$\mathtt{[}\mathcal{S}(E)\mathtt{]}\ \mathtt{[}\mathcal{S}(E^e)\mathtt{]}$' matches the description '$\mathtt{[}V_\alpha\mathtt{P}III|NNN\mathtt{(\%,}\ldots\mathtt{,\%,}V_p\mathtt{,\%,}\ldots\mathtt{,\%)}V_\omega\mathtt{]}\ \mathtt{[}v_\alpha v_p v_\omega\mathtt{]}$', where there are $i-1$ %-expressions before and $n-i$ %-expressions after the V_p. This corresponds exactly to the first object definition in section (3.3).

In the case that f is defined as a primitive recursion, we have again $E_s = f(s^{j_1}(0),\ldots,s^{j_n}(0))$. E_s^e now depends on the value of j_n. If $j_n = 0$, then according to the definition of *prfs*, $E_s^e = g(s^{j_1}(0),\ldots,s^{j_n-1})$. For $j_n > 0$, $E_s^e = h(s^{j_1}(0),\ldots,s^{j_n},f(s^{j_1}(0),\ldots,s^{j_n-1}))$. By the same argument as in the previous paragraph, the difference between $\mathcal{S}(E)$ and $\mathcal{S}(E^e)$ will only be the substitution of the substring $\mathcal{S}(E_s)$ by $\mathcal{S}(E_s^e)$. The string '$\mathtt{[}\mathcal{S}(E)\mathtt{]}\ \mathtt{[}\mathcal{S}(E^e)\mathtt{]}$' will therefore match

'$\mathtt{[}V_\alpha\mathtt{F(}V_1\mathtt{,}\ldots\mathtt{,}V_{n-1}\mathtt{,0)}V_\omega\mathtt{]}\ \mathtt{[}v_\alpha\mathtt{G(}v_1\mathtt{,}\ldots\mathtt{,}v_{n-1}\mathtt{)}v_\omega\mathtt{]}$', if $j_n = 0$, or
'$\mathtt{[}V_\alpha\mathtt{F(}V_1\mathtt{,}\ldots\mathtt{,}V_{n-1}\mathtt{,s}V_n\mathtt{)}V_\omega\mathtt{]}\ \mathtt{[}v_\alpha\mathtt{H(}v_1\mathtt{,}\ldots\mathtt{,}v_{n-1}\mathtt{,s}v_n\mathtt{,F(}v_1\mathtt{,}\ldots\mathtt{,}v_n\mathtt{)}v_\omega\mathtt{]}$', if $j_n > 0$.

These are dictionary objects as defined in section 3.3. □

Proof of theorem 2: The first parts of the dictionary objects all have the following form:

$$`[V_\alpha FNAME(arg_1,\ldots,arg_n)V_\omega]', \tag{5}$$

where $FNAME \in \mathcal{N}$ is a string representing a function name and the arg_i are strings '$\mathtt{s}^i\mathtt{0}$' with $i \geq 0$. The latter is due to the matching restrictions mentioned in section 3.3. Since $\mathcal{N}$ is suffix free, *FNAME* is uniquely determined by the match[5]. By assumption, $\mathcal{S}(E^e)$ is the string representation of a valid primitive recursive expression E. So the substring '$FNAME(arg_1,\ldots,arg_n)$' is the string representation of a subexpression E_s.

The theorem now follows immediately from the fact that all dictionary objects are constructed to have the form '$\mathtt{[}V_\alpha\mathcal{S}(E_s)V_\omega][v_\alpha\mathcal{S}(E_s^e)v_\omega\mathtt{]}$, where E_s^e can be derived from E_s by a strict evaluation step. E_s^e is the subexpression of the primitive recursive expression E^e with '$\mathtt{[}\mathcal{S}(E^e)\mathtt{]}$' = '$\mathtt{]}\ \mathtt{[}v_\alpha\mathcal{S}(E_s^e)v_\omega\mathtt{]}$' (the second part of the object) and thus the expression E^e can be derived from E by a strict evaluation step. □

[5]If $\mathcal{N}$ was not suffix free, e.g. there were the two function names '**add**' and '**dd**', a string like '**[add(0,0)]**' could be matched with '$\mathtt{[}V_\alpha\mathtt{add(}V_1\mathtt{,0)}V_\omega\mathtt{]}$' as well as with '$\mathtt{[}V_\alpha\mathtt{dd(}V_1\mathtt{,0)}V_\omega\mathtt{]}$'.

References

[1] Aho, A.V.: Algorithms for Finding Patterns in Strings. Handbook of Theoretical Computer Science, J. van Leeuwen, ed.. Elsevier Science Publishers B.V. (1990)

[2] Blasig, R.: Discrete Sequence Prediction with Commented Markov Models. To appear: International Congress on Grammatical Inference, Montpellier, France, 1996.

[3] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Mass. (1979)