Learning Multi-Valued Biological Models with Delayed Influence from Time-Series Observations

Tony Ribeiro
SOKENDAI (The Graduate University for Advanced Studies),
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
Email: tony_ribeiro@nii.ac.jp

Morgan Magnin
Institut de Recherche en Communications et
Cybernétique de Nantes (IRCCyN), École Centrale de Nantes,
1 rue de la Noë, 44321 Nantes, France
Email: morgan.magnin@irccyn.ec-nantes.fr

Katsumi Inoue
National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan,
Email: inoue@nii.ac.jp

Chiaki Sakama
Wakayama University,
Sakaedani, Wakayama 640-8510, Japan
Email: sakama@sys.wakayama-u.ac.jp

***Abstract*—Delayed effects are important in modeling biological systems, and timed Boolean networks have been proposed for such a framework. Yet it is not an easy task to design such Boolean models with delays precisely. Recently, an attempt to learn timed Boolean networks has been made in [1] in the framework of learning state transition rules from time-series data. However, this approach still has two limitations: (1) The maximum delay has to be given as input to the algorithm; (2) The possible value of each state is assumed to be Boolean, i.e., two-valued. In this paper, we extend the previous learning mechanism to overcome these limitations. We propose an algorithm to learn multi-valued biological models with delayed influence by automatically tuning the delay. The delay is determined so as to minimally explain the necessary influences. The merits of our approach is then verified on benchmarks coming from the DREAM4 challenge.**

***Keywords*—dynamical systems; multi-valued models; Boolean networks; delay; Markov(k); learning from interpretation transition; inductive logic programming**

I. Introduction

In some biological and physical phenomena, effects of actions or events appear at some later time points. For example, delayed influence can play a major role in various biological systems of crucial importance, like the mammalian circadian clock [2] or the DNA damage repair [3]. While Boolean networks have proven to be a simple, yet powerful, framework to model and analyze the dynamics of the above examples, they usually assume that the modification of one node results in an immediate activation (or inhibition) of its targeted nodes [4] for the sake of simplicity. But this hypothesis is sometimes too broad and we really need to capture the memory of the system i.e., keep track of the previous steps, to get a more realistic model. Our work aims to give an efficient and valuable approach to learn such dynamics.

The most used framework to model delayed and indirect influences in Boolean networks was designed by A. Silvescu et al. [5]: the authors introduced an extension of Boolean networks from a Markov(1) to Markov(k) model, where k is the number of time steps during which a variable can influence another variable. This extension is called temporal Boolean networks, abridged as $TBN(n, m, k)$, with n the number of variables and the expression of each variable at time $t+1$ being controlled by a Boolean function of the expression levels of at most m variables at times in $\{t, t-1, \ldots, t-(k-1)\}$. In this paper, we will consider Markov(k) model and discuss a new learning algorithm.

In some previous works, Markov(1) state transition systems are represented with logic programs [6], in which the state of the world is represented by an Herbrand interpretation and the dynamics that rule the environment changes are represented by a logic program P. The rules in P specify the next state of the world as an Herbrand interpretation through the *immediate consequence operator* (also called the T_P *operator*). With such a background, Inoue *et al.* [7] have recently proposed a framework to learn logic programs from traces of interpretation transitions (LFIT). The learning setting of this framework is as follows. We are given a set of pairs of Herbrand interpretations (I, J) as positive examples such that $J = T_P(I)$, and the goal is to induce a *normal logic program* (NLP) P that realizes the given transition relations.

We recently extended these researches by designing an algorithm that takes multiple sequences of state transition as input and builds a normal logic program that captures the delayed dynamics of a Markov(k) system [1], however limited to interacting components modeled as Boolean variables.

Boolean paradigm may appear as a simplified formalism, but it has led to significant results on the behavior of regulatory networks, particularly in terms of cycle behavior or steady states. Boolean values are however not sufficient to capture the complexity of some systems. For example, when a biological

component activates one gene and inhibits a third one, there is a very low probability that these interactions become effective at the same concentration level of the input component. The need for such multi-valued extension has been discussed and illustrated on different biological case studies, like immunity control in bacteriophage lambda [8] or p53-MdM2 network [3]. That is why Boolean modeling principles were extended so that the model can capture different levels (discrete) expression. This has opened the way to multi-valued logical modeling, as studied in many papers among the last twenty years [9].

In this paper, we aim to capture delayed influences in such multi-valued networks, which allow a more consistent representation of biological systems.

The paper is organized as follows: Section II introduces the logical background of this work. It also summarizes the main ideas behind the existing LF1T algorithm in order to make its extension to multi-valued models in Section III and IV be more understandable. We give and describe the new algorithm in Section V and discuss its evaluation in section VI. Finally, we conclude and discuss further works in Section VII.

II. Preliminaries

A. Logic Programming

In this section, we recall some preliminaries of logic programming. We consider a first-order language and denote the Herbrand base (the set of all ground atoms) as $\mathcal{B}$. A (*normal*) *logic program* (NLP) is a set of *rules* of the form

$$A \leftarrow A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n \qquad (1)$$

where A and A_i's are atoms ($n \geq m \geq 0$). For any rule R of the form (1), the atom A is called the *head* of R and is denoted as $h(R)$, and the conjunction to the right of $\leftarrow$ is called the *body* of R. We represent the set of literals in the body of R of the form (1) as $b(R) = \{A_1, \ldots, A_m, \neg A_{m+1}, \ldots, \neg A_n\}$, and the atoms appearing in the body of R positively and negatively as $b^+(R) = \{A_1, \ldots, A_m\}$ and $b^-(R) = \{A_{m+1}, \ldots, A_n\}$, respectively. When $b(R) = \emptyset$, the rule is called a *fact rule*.

An (*Herbrand*) *interpretation* I is a subset of $\mathcal{B}$. For a logic program P and an Herbrand interpretation I, the *immediate consequence operator* (or T_P *operator*) [10] is the mapping $T_P : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$:

$$T_P(I) = \{\, h(R) \mid R \in ground(P),\, b^+(R) \subseteq I,\, b^-(R) \cap I = \emptyset \}. \qquad (2)$$

B. Learning from Interpretation Transition

In this section, we recall the basis of learning from interpretation transition. *LF1T* [7] is an *any time algorithm* that takes a set of one-step state transitions E as input. These one-step state transitions can be considered as positive examples. From these transitions, the algorithm learns a logic program P that represents the dynamics of E. To perform this learning process, we can iteratively consider one-step transitions. In *LF1T*, the set of all atoms $\mathcal{B}$ is assumed to be finite. In the input E, a state transition is represented by a pair of interpretations (subset of $\mathcal{B}$). The output of *LF1T* is a logic program that realizes all state transitions of E.

Learning from 1-Step Transitions (LF1T)
Input: E a set of state transitions (I, J) and $\mathcal{B}$ the set of all possible atoms that can appear in I and J.
Output: A logic program P such that $J = T_P(I)$ holds for any $(I, J) \in E$.

In order to build a logic program with *LF1T*, we proposed in [11] a bottom-up method that generates hypotheses by *specialization* from the most general rules, that are fact rules, until the logic program is consistent with all input state transitions. Learning by specialization ensures to output the most general consistent rules, the prime rules (Def. 6 of [11]).

III. Multi-valued System

In this section we extend the formalization of [1] to handle multi-valued systems. The new algorithm can also compute the delay dynamically, whereas, in the previous version, the delay has to be given as an input, and was fixed. This algorithm can also be used to learn multi-valued Markov(1) system, thus it will guarantee to output only prime rules [11]. In order to represent multi-valued variables, we now restrict all atoms of a logic program to the form var^{val}. The intuition behind this form is that var represents some variable of the system and val represents the value of this variable. Our formalization of multi-valued logic program is based on annotated logics [12]. In annotated logics, the atom var is said to be annotated by the constant val. We consider a *multi-valued logic program* as a set of *rules* of the form

$$var^{val} \leftarrow var_1^{val_1} \wedge \cdots \wedge var_n^{val_n} \qquad (3)$$

where var^{val} and $var_i^{val_i}$'s are atoms ($n \geq 1$). For any rule R of the form (3), the atom var^{val} is called the *head* of R and is denoted as $h(R)$, and the conjunction to the right of $\leftarrow$ is called the *body* of R. We represent the set of literals in the body of R of the form (3) as $b(R) = \{var_1^{val_1}, \ldots, var_n^{val_n}\}$. A rule R of the form (3) is interpreted as follows: the variable var takes the value val in the next state if all variable var_i have the value val_i in the current state. An interpretation of a multi-valued program provides the value of each variable of the system and is defined as follows.

Definition 1 (Multi-valued interpretation): Let $\mathcal{B}$ be a set of atoms where each element has the form var^{val}. An *interpretation* I is a subset of $\mathcal{B}$ where $\forall var^{val} \in \mathcal{B}, \exists var^{val'} \in \mathcal{B}$ such that $var^{val'} \in I$ and $\forall var^{val'} \in I, \nexists var^{val''} \in I, val' \neq val''$ (In an interpretation each variable has one and only one value).

For a system S represented by a multi-valued logic program P and a state s_1 represented by an interpretation I, the successor of s_1 is represented by the interpretation:

$$next(I) = \{h(R) \mid R \in P, b(R) \subseteq I\}$$

The state transitions of a logic program P are represented by a set of pairs of multi-valued interpretations $(I, next(I))$.

Definition 2 (Multi-valued consistency): Let R be a rule and (I, J) be a state transition. R is *consistent* with (I, J) iff $b(R) \subseteq I$ implies $h(R) \in J$. Let E be a set of state transitions, R is consistent with E if R is consistent with all state transitions of E. A logic program P is *consistent* with E if all rules of P are *consistent* with E.

The notion of subsumption among rules is formally the same as for the Boolean case. We say that a rule R_1 is *more general* than another rule R_2 if $b(R_1) \subseteq b(R_2)$. In particular, a rule R is *most general* if there is no rule $R'(\neq R)$ that subsumes R ($b(R) = \emptyset$). To learn multi-valued logic programs with **LF1T** we need to adapt the least specialization of [11] to handle non-Boolean variables.

Definition 3 (Multi-valued least specialization): Let R_1 and R_2 be two rules such that $h(R_1) = h(R_2)$ and R_1 subsumes R_2. Let $\mathcal{B}$ be a set of atoms. The least specialization $ls(R_1, R_2, \mathcal{B})$ of R_1 over R_2 w.r.t $\mathcal{B}$ is $ls(R_1, R_2, \mathcal{B}) = \{h(R_1) \leftarrow b(R_1) \wedge var^{val'} | var^{val} \in b(R_2) \setminus b(R_1), var^{val'} \in \mathcal{B}, val' \neq val\}$

Example 1: Let $R_1 := a^1 \leftarrow a^1 \wedge b^2 \wedge c^1$ and $R_2 := a^1 \leftarrow a^1$ be two rules and let $\mathcal{B} = \{a^1, a^2, b^1, b^2, b^3, c^1, c^2\}$. The least specialization $ls(R_1, R_2, \mathcal{B})$ of R_1 over R_2 w.r.t $\mathcal{B}$ is $\{a^1 \leftarrow a^1 \wedge \mathbf{b^1}, a^1 \leftarrow a^1 \wedge \mathbf{b^3}, a^1 \leftarrow a^1 \wedge \mathbf{c^2}\}$ Here to avoid the subsumbtion of R_1 by R_2 according to $\mathcal{B}$, we can rather add a condition to R_2 over the variable b with another value than b^2 already in R_1, or a condition over the variable c other than c^1 already in R_1.

Least specialization can be used on a rule R to avoid the subsumption of another rule with a minimal reduction of the generality of R. By extension, least specialization can be used on the rules of a logic program P to avoid the subsumption of a rule with a minimal reduction of the generality of P. Let P be a logic program, $\mathcal{B}$ be a set of atoms, R be a rule and S be the set of all rules of P that subsume R. The least specialization $ls(P, R, \mathcal{B})$ of P by R w.r.t $\mathcal{B}$ is as follows:

$$ls(P, R, \mathcal{B}) = (P \setminus S) \cup (\bigcup_{R_P \in S} ls(R_P, R, \mathcal{B}))$$

IV. Multi-valued Markov(k) systems

In this section, we recall the formalization of [1] about Markov(k) systems and adapt it to multi-valued variables. A *Markov(k) system* can be seen as a k-steps deterministic system. In other words, the state of the system may depend on its (at most) k previous states. i.e., for any sequence of k state transitions there is only one possible state at time step $k+1$. If a system is Markov(k), it means that k is the maximum number of time steps such that the influence of any component (e.g., a gene) on another component is expressed. In other words, the state of a system may then depend on its (at most) k previous states.

Definition 4 (Timed Herbrand Base): Let P be a logic program. Let $\mathcal{B}$ be the Herbrand base of P and k be a natural number. The timed Herbrand Base of P (with period k) denoted by $\mathcal{B}_k$, is as follows:

$$\mathcal{B}_k = \bigcup_{i=1}^{k} \{var_{t-i}^{val} | var^{val} \in \mathcal{B}\}$$

where t is a constant term which represents the current time step.

According to Definition 4, given a propositional atom var^{val}, var_j^{val} is a new propositional atom for each $j = t - i$, $(0 \leq i \leq k)$. A Markov(k) system can then be interpreted as a logic program as follows.

Definition 5 (Markov(k) system): Let P be a logic program, $\mathcal{B}$ be the Herbrand base of P and $\mathcal{B}_k$ be the timed Herbrand base of P with period k. A Markov(k) system S with respect to P is a logic program where for all rules $R \in S$, $h(R) \in \mathcal{B}$ and all atoms appearing in $b(R)$ belong to $\mathcal{B}_k$.

In a Markov(k) system S, the atoms that appear in the body of the rules represent the value of the atoms that appear in the heads, but at previous time steps. In a context of modeling gene regulatory networks, these latter atoms represent the concentration of the interacting genes. This concentration is abstracted as an integer value modeling the fact that it is lower or greater than certain thresholds. Trace of executions, their consistency and k-step interpretations are formally equivalent to the Boolean case formalized in [1].

V. Algorithm

In [1] we proposed a method to learn delayed influences of Boolean systems: the *LFkT* algorithm. **LFkT** is an algorithm that can learn the dynamics of a Markov(k) system from its traces of execution. **LFkT** takes a set of traces of executions O as input, where each trace is a sequence of state transitions. If O is consistent, the algorithm outputs a logic program P that realizes all transitions of O. In this section, we propose a new version of this algorithm that handle multi-valued variables. Furthermore, the delays are now computed dynamically and do not need to be known or fixed to the size of the longest trace.

LFkT:

Input: A set of traces of executions O of a multi-valued Markov(k) system S.
Step 1: Initialize a logic program with fact rules.
Step 2: Pick a trace T from O and update the delay considered accordingly.

- Initialize a logic program with fact rules for each new delay.
- Revise these logic programs with all previous traces (like step 3).

Step 3: Convert the trace into interpretation transitions and revise the logic programs using least specialization.
Step 4: If there is remaining trace in O, go back to step 2.
Step 5: Merge all logic programs into one while avoiding rules subsumption.
Step 6: Remove all rules that are not necessary to explain the observations.
Output: A set of rules which realizes O.

The detailed pseudo code of **LFkT** is given in Algorithm 1.

1) The algorithm starts with a logic program that only contains all possible fact rules and assumes that the system to learn is Markov(1) (lines 6-9). These different programs are merged at the end to constitute a logic program that realizes all consistent traces of O. **2.1)** Before learning from a trace, we need to guarantee that we are considering a valid delay according to the trace (lines 13-20). That is why we check the minimal delay required to explain the trace by using the **delay** function, whose pseudo code is given in Algorithm 2. If this delay is greater than the one currently considered by the algorithm, it updates this delay and generates programs

Algorithm 1 **LFkT**(O,$\mathcal{B}$) : Learn a set of rules that realize O

INPUT: O a set of traces of executions, $\mathcal{B}$ a set of atoms
OUTPUT: P a logic program that realizes the transitions of O.
P' a vector of set of rules
E a set of pairs of interpretations (I, J)
k an integer
// 1) Initialize P' with the most general logic program
for each atom $var^{val} \in \mathcal{B}$ **do**
    $P'_1 := P'_1 \cup \{var^{val} \leftarrow\}$
$k := 1$ // Assume Markov(1)
// 2) Learning phase
while $O \neq \emptyset$ **do**
    pick a trace $T \in O$
    // 2.1) Check delay of the trace
    if **delay**$(T) > k$ **then** // Extend the delay to learn
        for $i = k + 1$ to $delay(T)$ **do**
            for each atom $var^{val} \in \mathcal{B}$ **do**
                for each atom $var'^{val'} \in \mathcal{B}$ **do**
                    $P'_i := P'_i \cup \{var^{val} \leftarrow var'^{val'}_{t-i}\}$
        $k :=$ **delay**(T)
        for each trace $T' \in O'$ **do**
            $P' := \mathbf{learn}(P, T', k, \mathcal{B})$
    // 2.2) Check consistency with previous traces
    if $\exists T' \in O', T$ and T' are not k-consistent **then**
        for each k' from k to $min(|T|, |T'|)$ **do**
            if T and T' are k'-consistent **then**
                for $i = k$ to k' **do**
                    for each atom $var^{val} \in \mathcal{B}$ **do**
                        for each atom $var'^{val'} \in \mathcal{B}$ **do**
                            $P'_i := P'_i \cup \{var^{val} \leftarrow var'^{val'}_{t-i}\}$
                for each trace $T' \in O'$ **do**
                    $P' := \mathbf{learn}(P, T', k, \mathcal{B})$
                k := k'
            else//T and T' are not consistent, cannot happen if O is consistent
                EXIT: non-deterministic input
    // 2.3) Specify P' by the interpretations of the trace
    $P' := \mathbf{learn}(P, T, 1, \mathcal{B})$
    $O := O \setminus \{T\}$
    $O' := O' \cup \{T\}$
end while
// 3) Merge the programs into a unique logic program
$merging := \emptyset$
for each i from 1 to k **do**
    remove from P'_i all rules subsumed by a rule of $merging$
    $merging := merging \cup P'_i$
// 4) Keep only the rules that can realize the observations
$P := \emptyset$
for each $T' \in O'$ **do**
    $E :=$ **interprete**(T')
    for each $(I, J) \in E$ **do**
        for each $R \in merging$ **do**
            if $b(R) \subseteq I$ and $h(R) \in J$ **then**
                $P := P \cup \{R\}$
return P

Algorithm 2 **delay**(T) : Compute the minimal delay of a trace

INPUT: a trace of execution $T = (S_0, \ldots, S_n)$
OUTPUT: $delay$ an integer

delay := 1
for each i from 1 to $n - 1$ **do**
    for each j from i to $n - 1$ **do**
        if $S_i = S_j$ **then**
            k := 1
            while $k \leq i$ AND $S_{i-k} = S_{j-k}$ **do**
                $k := k + 1$
            end while
            delay := max(delay,k)
return $delay$

for all missing delays (lines 14-20). All previously analyzed traces are then re-analyzed but only for these new programs. This allows to learn only the missing delayed rules. **2.2)** Then it checks the consistency of the new trace with previously

Algorithm 3 **learn**$(P, T, min\_delay, \mathcal{B})$: Revise P to avoid the subsumption of R

INPUT: P a vector of logic program, T a trace of execution and min_delay an integer
OUTPUT: a vector of logic program

$E :=$ **interprete**(T)
for each i from min_delay to $|T|$ **do**
    for each k-step interpretation $(I, J) \in E$ with $k \geq i$ **do**
        remove from I all atoms var^{val}_{t-n} with $n > i$
        for each atom $var^{val} \in J$ **do**
            for each $var^{val'} \in \mathcal{B}, val' \neq val$ **do**
                $R^I_{var^{val'}} := var^{val'} \leftarrow \bigwedge_{l_j \in I} l_j$
                $P_i :=$ **Specialize**$(P_i, R^I_{var^{val'}}, \mathcal{B})$
return P

analyzed ones (lines 21-31). The delay considered is increased if necessary. In practice, the consistency of the new traces with previously analyzed ones can be directly checked from the programs that are learned. If the program that considers the biggest delay k has no rule that can realize the last transition of T (if $\nexists R \in P'_k, b(R) \subseteq I$ with $(I, S_n) :=$ the $|T|$-step interpretation transition of T), then the trace is not k-consistent with at least one of the previous ones. **2.3)** The program that is learned is revised according to the new trace using least specialization (lines 34-37). In order to use least specialization, we need to convert the trace of execution into interpretation transitions. This conversion is done by the function **interprete**, whose pseudo code is given in Algorithm 4. Here, $min(k, |T|)$ interpretation transitions are extracted from the trace, one for each possible delay inferior to the currently considered one, that is k. Following this method, it produces one $min(k, |T|)$-step interpretation, one $min(k, |T|) - 1$ interpretation, ..., one 1-step interpretation. The function outputs them as a vector of interpretation transitions E, where each E_i corresponds to an i-step interpretation transition of a sub-trace of size i of T. The

Algorithm 4 **interprete**(T) : Extract interpretation transitions from a trace

INPUT: a trace of execution $T = (S_0, \ldots, S_n)$
OUTPUT: E a set of pairs of interpretations

$E := \emptyset$
// Extract interpretations
for each k from 1 to $|T|$ **do**
    $T' := (S_0, \ldots, S_k)$ // the sub-trace of size k of T that start from S_0
    $I := \emptyset$
    for each state $s_{k'}$ before s_k in T' **do**
        $delay := k - k'$
        for each atom $a \in s_{k'}$ **do**
            $I := I \cup \{a_{t-delay}\}$
    $E := E \cup (I, S_k)$
return E

algorithm iteratively learns from each pair of interpretations of E. Now it only needs to apply the least specialization by analyzing each pair of interpretations $(I, J) \in E$. For each atom var^{val} that **does not appear** in J, it infers an **anti-rule**: $R^I_{var^{val}} := var^{val} \leftarrow \bigwedge_{B_i \in I} B_i$, Then, least specialization is used to make each corresponding logic program P'_i consistent with $R^I_{var^{val}}$, according to the delay of interpretation transition. Algorithm 5 shows the pseudo code of this operation. In the function **specialize**, it first extracts all rules $R_P \in P$ that subsumes R^I_A. It generates the least specialization of each R_P by generating a rule for each literal in $R^I_{var^{val}}$. Each rule contain all literals of R_P, plus a literal that represents another

Algorithm 5 **specialize**(P,R,$\mathcal{B}$) : specialize P to avoid the subsumption of R

INPUT: a logic program P, a rule R, a set of atoms B
OUTPUT: the least specialization of P by R.

$conflicts$: a set of rules
$conflicts := \emptyset$
// Search rules that need to be specialized
for each rule $R_P \in P$ **do**
  if R_P subsumes R **then**
    $conflicts := conflicts \cup R_P$
    $P := P \setminus R_P$
// Revise the rules by least specialization
for each rule $R_c \in conflicts$ **do**
  for each literal $var_{t-k}^{val} \in b(R)$ **do**
    if $var_{t-k}^{val} \notin b(R_c)$ **then**
      for each $var_{t-k}^{val'} \in \mathcal{B}, val' \neq val$ **do**
        $R'_c := (h(R_c) \leftarrow (b(R_c) \cup var_{t-k}^{val'}))$
        if P does not subsume R'_c **then**
          $P := P \setminus$ all rules subsumed by R'_c
          $P := P \cup R'_c$
return P

value of the variable represented by a literal in $R^I_{var^{val}}$, so that $R^I_{var^{val}}$ is not subsumed anymore by that rule. Then **specialize** adds in P all the generated rules that are not subsumed by P, so that P becomes consistent with the transition (I, J).

3) After analyzing all traces of O, the k programs that have been learned are merged into a unique logic program while taking care that subsumed rules are discarded. **4)** All rules that are not necessary to explain the observations are discarded. The algorithm only keeps the rules that can be used to realize at least one of the transition of the input traces. Finally, **LFkT** outputs a logic program that realizes all consistent traces of execution of O.

Theorem 1 (Correctness of **LFkT***):* Let P be a logic program, $\mathcal{B}$ be the Herbrand base of P and $\mathcal{B}_k$ be the timed Herbrand base of P with period k. Let S be a Markov(k) system with respect to P. Let O be a set of traces of S. Using O as input, **LFkT** outputs a logic program that realizes all consistent traces of O. Proof is given in appendix.

Theorem 2 (Complexity): Let P be a logic program, $\mathcal{B}$ be the Herbrand base of P and $\mathcal{B}_k$ be the timed Herbrand base of P with period k. Let S be a multi-valued Markov(k) system with respect to P. Let n be the number of variable of S. Let v be the maximal number of value of a variable of S. Let O be a set of traces of execution of S. The complexity of learning S from O with **LFkT** is respectively: $O(n \cdot v^{nk+1} + |O|)$ for memory and $O(\sum_{T \in O} |T| \cdot nv^{nk+3} + |O| \cdot n^2k^2 + n \cdot v^{nk+2} + n \cdot v^{nk+1} \cdot |O| \cdot k)$ for runtime. Proof is given in appendix.

VI. Evaluation

In this section, we assess the efficiency of our new **LFkT** algorithm through case studies coming from the DREAM4 challenge [13].

DREAM challenges are annual reverse engineering challenges that provide biological case studies. In this paper, we focus on the datasets coming from DREAM4. The input data that we tackle here consists of the following: 5 different systems each composed of 10 genes, all coming from E. coli and yeast networks. For every such system, the available data are the following: (i) 5 time series data with 21 time points; (ii) steady state at wild type; (iii) steady states after knocking out each gene; (iv) steady states after knocking down each gene (i.e. forcing its transcription rate at 50%); (v) steady states after some random multifactorial perturbations. We processed all the data. Because of the lack of space, we focus here on the management of time series data.

Benchmark	run time	raw output	final output	Mean squared error
insilico_size10_1	28s	118,834	359	0.073
insilico_size10_2	2m5s	401,923	462	0.064
insilico_size10_3	44s	151,021	480	0.019
insilico_size10_4	22s	90,904	387	0.031
insilico_size10_5	1m04s	297,364	326	0.091

TABLE I. Evaluation of **LFkT** on learning and prediction of gene regulatory network benchmarks from the DREAM4 challenge.

A. Settings

Time series data provide us 20 transitions. Each of them include different perturbations that are maintained all time along during the first 10 transitions and applied to at most 3 genes. In this setting, a perturbation means a significant increase or decrease of the gene expression. In the raw data of the time series, gene expression values are given as real number between 0 and 1. To apply our approach, we chose to discretize those data into 4 qualitative values. Each gene is discretized in an independent manner, with respect to the following procedure: we compute the average value of the gene expression among all data of a time series, then the values between the average and the maximal/minimal value are divided into as many levels. Discretizing the data according to the average value of expression is expected to reduce the impact of perturbation on the discretization and thus on the rules that are learned.

B. Results

Table I shows the evolution of runtime, output size and precision of prediction of **LFkT** on the five benchmarks of the DREAM4 challenge. Here, we use **LFkT** to learn rules independently for each time series. The rules learned on one time series are evaluated on the others series in a cross-validation manner. The precision of each rule is computed as the ratio between the number of times they match a transition and how many transitions they realize. For a transition (I, J) when $b(R) \subseteq I$, the rule R matches the transition and if $h(R) \in J$, R realizes the transition. To use the model learned to predict the next state of the system we simply apply the rules with the best ratio that matches the current state. By doing so, we expect to reduce the impact of the perturbations on the predictions of the model learned.

The number of generated rules is huge, but the following simple heuristics can be used to greatly reduce it. First, rules that never realize any transitions can be discarded. Rules that are subsumed by rules with a better or equal precision are discarded. This allows to remove about 50% of the rule generated. We can also detect and discard rules that will never be used for prediction: the rules that never have a better precision than an other rule that matches a same state with a different conclusion. This allows to remove about 99% of the remaining rules. Using those simple heuristics does not impact the dynamic of the model that is learned: the prediction will be

exactly the same. But it simplifies the model that is learned and makes it more human readable. The run time showed in Table I includes: learning, cross-validation and applications of the heuristics. All experiments are run with a C++ implementation of **LFkT** on a processor Intel Xeon (X5650, 2.67GHz) with 12GB of RAM.

The DREAM4 challenge offers two different problems, which consist in predicting (i) the structure of the gene interactions (in terms of an unsigned directed graph); (ii) attractors in some given conditions. So far, **LFkT** is not designed to efficiently tackle the first issue. Indeed, **LFIT** method focuses on the learning of the dynamics of the observed system. This means that the rules learned by **LFkT** will be recurrent patterns, i.e., correlations between the evolution of values of the different variables. This method however can be fully applied to predict attractors. For this evaluation, we are given an initial state and 5 different dual gene knockouts conditions. The goal is to predict the attractor in which the system will fall from the initial state for each dual knockout. In the challenge, the quality of the prediction is evaluated by computing the mean square error between the predicted state and the expected one. The precision we achieved in those experiments is quite good considering the results of the competitors of the DREAM4 challenge [14]. Their results range between 0.01 and 0.075 for the same evaluation settings, which we are comparable to.

We plan to pursue these evaluations. Competitors of DREAM4 Challenge tackled not only the networks with 10 genes, but also the ones consisting of 100 genes with no drastic loss of precision. In this on-going work, we now consider applying our approach to these large networks, to get stronger arguments in favor of our approach. Currently, the implementation of our algorithm is too greedy in terms of memory to handle those networks. We need several technical optimizations and heuristic to be able to tackle those problem in practice. Furthermore, [14] showed much better prediction results (0.01 to 0.025), when changing the given initial state for the one based on single gene knockout. We should also consider to discuss this issue in future works in order to improve our method, which obtained encouraging first results as showed in this subsection.

VII. Conclusion

In this paper, we propose a twofold extension of our previous results to learn normal logic programs from interpretation transitions on k-steps: (i) Delay is now dynamically adjusted and does not need to be initally assumed as input.; (ii) The learning algorithm natively tackles multi-valued models. The work can then be directly applied to the learning of Boolean and multi-valued discrete networks with delayed influences, which is crucial to understand the memory effect involved in some interactions between biological components. Further works aim at adapting the approach developed in the paper to the kind of data as produced by biologists [15]. This requires to connect through various databases in order to extract real time series data, and subsequently explore and use them to learn genetic regulatory networks. We also consider extending the methodology to asynchronous semantics, which can help to capture more realistic behaviors.

References

[1] T. Ribeiro, M. Magnin, K. Inoue, and C. Sakama, "Learning delayed influences of biological systems," Frontiers in Bioengineering and Biotechnology, vol. 2, p. 81, 2015.

[2] J.-P. Comet, G. Bernot, A. Das, F. Diener, C. Massot, and A. Cessieux, "Simplified models for the mammalian circadian clock," Procedia Computer Science, vol. 11, pp. 127–138, 2012.

[3] W. Abou-Jaoudé, D. A. Ouattara, and M. Kaufman, "From structure to dynamics: frequency tuning in the p53–mdm2 network: I. logical approach," Journal of theoretical biology, vol. 258, no. 4, pp. 561–577, 2009.

[4] T. Akutsu, S. Kuhara, O. Maruyama, and S. Miyano, "Identification of genetic networks by strategic gene disruptions and gene overexpressions under a boolean model," Theoretical Computer Science, vol. 298, no. 1, pp. 235–251, 2003.

[5] A. Silvescu and V. Honavar, "Temporal boolean network models of genetic networks and their inference from gene expression time series," Complex Systems, vol. 13, no. 1, pp. 61–78, 2001.

[6] K. Inoue, "Logic programming for boolean networks," in IJCAI Proceedings-International Joint Conference on Artificial Intelligence, vol. 22, no. 1. Citeseer, 2011, p. 924.

[7] K. Inoue, T. Ribeiro, and C. Sakama, "Learning from interpretation transition," Machine Learning, vol. 94, no. 1, pp. 51–79, 2014.

[8] D. Thieffry and R. Thomas, "Dynamical behaviour of biological regulatory networks-ii. immunity control in bacteriophage lambda," Bulletin of Mathematical Biology, vol. 57, no. 2, pp. 277–297, 1995.

[9] C. Chaouiya, A. Naldi, E. Remy, and D. Thieffry, "Petri net representation of multi-valued logical regulatory graphs," Natural Computing, vol. 10, no. 2, pp. 727–750, 2011.

[10] K. R. Apt, H. A. Blair, and A. Walker, "Foundations of deductive databases and logic programming," J. Minker, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, ch. Towards a Theory of Declarative Knowledge, pp. 89–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=61352.61354

[11] T. Ribeiro and K. Inoue, "Learning prime implicant conditions from interpretation transition," in The 24th International Conference on Inductive Logic Programming, 2014, to appear (long paper) (http://tony.research.free.fr/paper/ILP2014long.pdf).

[12] H. A. Blair and V. Subrahmanian, "Paraconsistent logic programming," Theoretical computer science, vol. 68, no. 2, pp. 135–154, 1989.

[13] R. J. Prill, J. Saez-Rodriguez, L. G. Alexopoulos, P. K. Sorger, and G. Stolovitzky, "Crowdsourcing network inference: the dream predictive signaling network challenge," Science signaling, vol. 4, no. 189, p. mr7, 2011.

[14] A. Greenfield, A. Madar, H. Ostrer, and R. Bonneau, "Dream4: Combining genetic and dynamic information to identify biological networks and dynamical models," PloS one, vol. 5, no. 10, pp. e13 397–e13 397, 2010.

[15] X. Li, S. Rao, W. Jiang, C. Li, Y. Xiao, Z. Guo, Q. Zhang, L. Wang, L. Du, J. Li et al., "Discovery of time-delayed gene regulatory networks based on temporal gene expression profiling," BMC bioinformatics, vol. 7, no. 1, p. 26, 2006.

APPENDIX

A. Proof of Theorem 1 (Correctness of **LFkT**)

Let P be a logic program, $\mathcal{B}$ be the Herbrand base of P and $\mathcal{B}_k$ be the timed Herbrand base of P with period k. Let S be a Markov(k) system with respect to P. Let O be a set of trace of S. Using O as input, **LFkT** output a logic program that realizes all consistent traces of O.

Proof: Let V be the vector of interpretation transition extracted from O by **LFkT** (Algorithm 4). According to Theorem 4 of [11], initializing *LF1T* with $\{p.|p \in \mathcal{B}\}$, by using minimal specialization iteratively on a set of interpretation transitions E, we obtain a logic program P that realizes E. Since **LFkT** uses this method on each element of V, **LFkT** learns a vector of logic programs P' such that each logic program $p'_n \in P'$ realizes the corresponding set of interpretation transitions $v_n \in V$, $n \geq 1$.

Let $p'_n \in P'$ be the logic program learn from $v_n \in V$, $n \geq 1$. p'_n is obtained by minimal specialization of $\{p.|p \in \mathcal{B}\}$ with all anti-rule of v_n (non consistent rule). According to Theorem 3 of [11], p'_n does not subsume any anti-rule that can be inferred from v_n. Then, p'_n realizes all deterministic transition of v_n, that is $\forall (I,J) \in v_n, \nexists (I,J'), J \neq J'$.

Since v_n contains n-step interpretation transition that represent all sub-traces of size n of O, p'_n realizes all consistent sub-trace of size n of O. Let P_{n-1} be a logic program that realizes all consistent sub-traces of size at most $n-1$ of O. p'_n can contain a rule R such that $(\mathcal{B}_n \setminus \mathcal{B}_{n-1}) \cap b(R) = \emptyset$ (no literal of R refers to the $t-n$ state of the variables). In this case R realizes a sub-trace of size n and also some sub-traces of size at most $n-1$. If these sub-traces of size $n-1$ are consistent, then they are necessary realized by P_{n-1}. $P_{n-1} \cup \{R\}$ does not realize more consistent sub-trace of size at most $n-1$ than P_{n-1}. Let S_R be the set of rules of p'_n of the form R, then $(p'_n \setminus S_R)$ only realizes all sub-traces of size n of O. Then the logic program $P_n = P_{n-1} \cup (p'_n \setminus S_R)$ only realizes all consistent sub-trace of size at most $n-1$ of O and all sub-traces of size n of O, that is P_n realizes all consistent sub-traces of size at most n of O.

Let $p'_1 \in P'$ be the logic program learned from $v_1 \in V$, and let $P = p'_1$. Let R' be all rules of the logic program p'_n such that $(\mathcal{B}_n \setminus \mathcal{B}_{n-1}) \cap b(R') \neq \emptyset$. Iteratively adding rules R' into P, starting by the logic program p'_2 until p'_k, we obtain a logic program that realizes all consistent sub-traces of size at most k of O. As a result, using O as input, **LFkT** outputs a logic program that realizes all consistent traces of O.

■

B. Proof of Theorem 2 (Complexity of **LFkT**)

Let P be a logic program, $\mathcal{B}$ be the Herbrand base of P and $\mathcal{B}_k$ be the timed Herbrand base of P with period k. Let S be a multi-valued Markov(k) system with respect to P. Let n be the number of variables of S. Let v be the maximal number of values of a variable of S. Let O be a set of traces of execution of S. The complexity of learning S from O with **LFkT** is respectively: $O(n \cdot v^{nk+1} + |O|)$ for memory and $O(\sum_{T \in O} |T| \cdot nv^{nk+3} + |O| * n^2k^2 + n \cdot v^{nk+2} + n \cdot v^{nk+1} \cdot |O| \cdot k)$ for runtime.

Proof: n is the number of possible heads of rules of S. nk is the maximum size of a rule of S, i.e. the number of literals in the body; a literal can appear at most one time in the body of a rule. For each rule head of $\mathcal{B}$ there are v^{nk} possible bodies: each literal can be present or absent from the body. From these preliminaries we conclude that the size of a Markov(k) system S learned by **LFkT** is at most $|S| = n \cdot v^{nk}$. To learn S, **LFkT** needs to store k programs P_i that are Markov(i) system with respect to P, $1 \leq i \leq k$. The algorithm also needs to store the previously analyzed traces in order to update the considered delay.

Conclusion 1: the memory use of **LFkT** is $O(\sum_{i=1}^{k} |P_i| + O) = O(k \cdot \frac{n \cdot v^{nk}}{k} + |O|)$ that is bound by $O(n \cdots v^{nk+1} + |O|)$.

For each trace T of O, **LFkT** extracts $|T|$ pairs of interpretations. For each pair of interpretation (I,J), **LFkT** infers an anti-rule rule R^I_A for each $A \in \mathcal{B}, A \notin J$. **LFkT** compares each R^I_A with all rules of each programs P_i. There is at most $|\mathcal{B}| - n$ anti-rules that can be infered from (I,J) by **LFkT** and the size of each program P_i is bound by $O(\frac{n \cdot v^{nk}}{k})$. Then, the complexity of learning one trace of execution $T \in O$ with **LFkT** is $O(|T| \cdot |\mathcal{B}| - n \cdot k |P_i|) = O(|T| \cdot nv - n \cdot k \frac{n \cdot v^{nk}}{k}) = O(|T| \cdot n^2 v^{nk+2} - n)$ that is bound by $O(|T| \cdot nv^{nk+3})$. To update the considered delay, the algorithm has to check the delay of each new trace T, this operation belongs to $O(|T|^2) = (n^2k^2)$. And, checking the consistency of a new traces with previous analyzed ones is bound by $O(|O| * n^2k^2)$. Merging the programs requires to compare all rules to detect subsumption, it has a complexity of $O(n \cdot v^{nk+2})$. Finally, removing the rules that are not necessary to realize O requires to compare each rule with all k-step interpretations of O, thus it requires $O(n \cdot v^{nk+1} \cdot |O| \cdot k)$.

Conclusion 2: The complexity of learning S from O with **LFkT** is $O(\sum_{T \in O} |T| \cdot nv^{nk+3} + |O| \cdot n^2k^2 + n \cdot v^{nk+2})$.

■