

DOCTORAL THESIS

Studies on Learning Dynamics of Systems from State Transitions

Author:

Tony RIBEIRO

Supervisor:

Prof. Katsumi INOUE

*A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy*

in the

Department of Informatics,
School of Multidisciplinary Sciences,
The Graduate University for Advanced Studies

June 2015

Abstract

In recent years, there has been a notable interest in the field of Inductive Logic Programming to learn from state transitions as part of a wider interest in learning the dynamics of systems. Learning system dynamics from the observation of state transitions has many applications in multi-agent systems, robotics and bioinformatics alike. Knowing the dynamics of its environment can allow an agent to identify the consequences of its actions more precisely. This knowledge can be used by agents and robots for planning and scheduling. In bioinformatics, learning the dynamics of biological systems can correspond to the identification of the influence of genes and can help the understanding of their interactions.

In this thesis, we study a method called learning from interpretation transition. The purpose of this method is to automatically construct a model of the dynamics of a system from the observation of its state transitions. In this method the dynamics of a system is represented by a logic program that is a set of transitions rules. The learning settings can be summarized as follows. We are given a set of state transitions and the goal is to induce a logic program that realizes the given transition relations.

In the first chapter we recall the background of the three main fields of research which our contribution belongs to, that are: Machine Learning, Logic Programming and Inductive Logic Programming. In the second chapter we introduce the preliminaries notions needed for the understanding of our contribution.

In the third chapter we introduce the basis of our framework for learning dynamics of system from state transition. We firstly tackle this induction problem by learning from synchronous state transitions. Given any Boolean state transitions diagram, we propose an algorithm that can learn a logic program that exactly captures the system dynamics. Then, we focus on the minimality of the rules learned. Our goal is to learn all minimal conditions that imply a variable to be true in the next state. In bioinformatics, for a gene regulatory network, it corresponds to all minimal conditions for a gene to be activated/inhibited. For this purpose, we propose another version of our algorithm which guarantees that all rules learned are minimal. In the fourth chapter, we provide several extensions of our framework. Here, we start to consider how our framework can contribute to model more complex Biological system. In some biological and physical phenomena, effects of actions or events appear at some later time points. We extend our framework by designing an algorithm that builds a logic program which captures the delayed dynamics of a system. So far, the systems that our algorithms could handle were restricted to Boolean variables. Boolean values are not sufficient to capture the complexity of some systems. That's why we extend our algorithms to handle multi-valued networks. Finally, the last contribution of this thesis is to learn dynamics of non-deterministic systems. We extend our framework to learn probabilistic dynamics by proposing an algorithm for learning from uncertain state transitions. This algorithm learns the probability of the value change of each variable of the system.

In the fifth chapter, we discuss related work and compare our approaches to others. Here we point out similarities and differences, assess advantages and weak points of the method we propose. Finally, in the last chapter, we conclude the thesis by summing up what have been done and discuss possible future works and perspectives.

Contents

| | |
|--|-----------|
| Abstract | i |
| Contents | ii |
| List of Figures | v |
| List of Tables | vi |
| 1 Introduction | 1 |
| 1.1 Background | 6 |
| 1.1.1 Machine Learning | 6 |
| 1.1.1.1 Supervised learning | 6 |
| 1.1.1.2 Unsupervised learning | 7 |
| 1.1.1.3 Reinforcement Learning | 8 |
| 1.1.2 Logic Programming | 9 |
| 1.1.2.1 Probabilistic Logic Programming | 10 |
| 1.1.3 Inductive Logic Programming | 12 |
| 2 Preliminaries | 13 |
| 2.1 Logic Programming | 14 |
| 2.2 Boolean network | 16 |
| 2.3 Representing Dynamics in Logic Programs | 17 |
| 2.4 Contribution | 20 |
| 2.4.1 First Steps | 20 |
| 2.4.2 Getting Stronger | 20 |
| 2.4.3 The Quest of Minimality | 21 |
| 2.4.4 Facing Delays | 22 |
| 2.4.5 Uncertain Future | 22 |
| 3 Learning From Interpretation Transitions | 24 |
| 3.1 Learning from 1-step Transitions | 25 |
| 3.1.1 Generalization by Naïve Resolution | 26 |
| 3.1.2 Generalization by Ground Resolution | 29 |
| 3.1.3 Experiments | 33 |
| 3.1.3.1 Learning Boolean Networks | 33 |
| 3.1.3.2 Learning big benchmark from partial state transitions sets | 34 |
| 3.1.3.3 Learning Cellular Automata | 36 |

| | | |
|----------|---|-----------|
| 3.1.4 | Conclusion | 41 |
| 3.2 | BDD Algorithms for LF1T | 43 |
| 3.2.1 | Algorithm | 45 |
| 3.2.2 | Evaluation | 52 |
| 3.2.3 | Conclusion | 54 |
| 3.3 | Learning Prime Implicant Conditions | 55 |
| 3.3.1 | Formalization | 55 |
| 3.3.2 | Learning with full Naïve/ground resolution | 56 |
| 3.3.3 | Least Specialization for LF1T | 57 |
| 3.3.4 | Algorithm | 59 |
| 3.3.5 | Evaluation | 65 |
| 3.4 | Conclusion and Future Work | 67 |
| 4 | Framework Extensions | 68 |
| 4.1 | Delayed Systems | 69 |
| 4.1.1 | Formalization | 69 |
| 4.1.2 | Algorithm | 72 |
| 4.1.3 | Running example | 77 |
| 4.1.4 | Evaluation | 79 |
| 4.1.5 | Conclusion | 81 |
| 4.2 | Multivalued Variables | 82 |
| 4.2.1 | Formalization | 82 |
| 4.2.2 | Algorithm | 84 |
| 4.3 | Multivalued Delayed Systems | 86 |
| 4.3.1 | Formalization | 86 |
| 4.3.2 | Algorithm | 87 |
| 4.3.3 | Running example of LFkT | 93 |
| 4.3.4 | Evaluation | 95 |
| 4.3.5 | Conclusion | 97 |
| 4.4 | Asynchronous Systems | 98 |
| 4.4.1 | Time delays in asynchronous framework | 99 |
| 4.4.2 | Formalization | 99 |
| 4.4.3 | Algorithms | 100 |
| 4.4.3.1 | Learning using generalization | 100 |
| 4.4.3.2 | Learning using specialization | 102 |
| 4.5 | Uncertainty | 105 |
| 4.5.1 | Formalization | 105 |
| 4.5.2 | Algorithm | 106 |
| 4.5.3 | Learning Probabilistic Action Models | 109 |
| 4.5.3.1 | Integration of Logic Programming and Planning Domains | 110 |
| 4.5.3.2 | Planning Model | 110 |
| 4.5.3.3 | Data Representation | 111 |
| 4.5.3.4 | Selecting the Set of Planning Operators | 112 |
| 4.5.3.5 | Planning Operators Requirements | 112 |
| 4.5.3.6 | Score Function | 113 |
| 4.5.3.7 | Selecting the Best Planning Operator Subset | 114 |
| 4.5.3.8 | Experiments | 115 |

| | | |
|----------|--|------------|
| 4.5.4 | Conclusions | 116 |
| 5 | Related Works and Comparaison | 117 |
| 5.1 | Related Work | 118 |
| 5.1.1 | Learning from Interpretations | 118 |
| 5.1.2 | Learning Probabilistic Logic Program | 118 |
| 5.1.3 | Learning Action Theories | 119 |
| 5.1.4 | Reinforcement Learning | 119 |
| 5.1.5 | Learning Nonmonotonic Programs | 120 |
| 5.1.6 | Learning Neural Networks | 120 |
| 5.1.7 | Learning Boolean Networks | 121 |
| 5.1.8 | Learning Petri Nets | 122 |
| 5.2 | Comparison | 123 |
| 5.2.1 | Computational learning theory | 123 |
| 5.2.2 | Learning from interpretations | 123 |
| 5.2.3 | Learning action theories | 124 |
| 5.2.4 | Reinforcement learning | 125 |
| 5.2.5 | Learning Nonmonotonic Programs | 125 |
| 5.2.6 | Learning Probabilistic Logic Program | 127 |
| 5.2.7 | Learning Boolean Networks | 127 |
| 5.2.8 | Learning Petri Nets | 129 |
| 5.2.9 | Learning Cellular Automata | 129 |
| 5.2.10 | Binary Decision Diagram | 129 |
| 5.2.11 | Inverse Ingeniering | 129 |
| 6 | Conclusions and Future Work | 131 |
| 6.1 | Summary of contribution | 131 |
| 6.2 | Perspectives | 132 |
| | Bibliography | 134 |

List of Figures

| | | |
|-----|---|-----|
| 1.1 | Big picture of our contribution | 1 |
| 1.2 | The four characters of our system: Blue, Brown and Yellow invited by Pink. . . | 3 |
| 1.3 | Trace of the chat discussion of the characters. | 4 |
| 1.4 | Discretization of the chat discussion into state transitions. | 4 |
| 2.1 | A Boolean network B_1 (left) and its state transition diagram (right) | 16 |
| 3.1 | State changes by Wolfram's Rule 110 | 37 |
| 3.2 | State changes by Wolfram's Rule 110 in Torus world | 38 |
| 3.3 | Evolution of the BDD of p in Example 3.2 | 44 |
| 4.1 | Eight traces of executions of the system of Example 4.1 | 71 |
| 4.2 | LFkT Run time varying the input size (number of traces) | 79 |
| 4.3 | Interaction graph and state transitions diagram of the system of Example 4.6 . . | 87 |
| 4.4 | Eight traces of executions of the system of Example 4.6 | 87 |
| 4.5 | Runtime and output size of LFkT varying the delay of the system | 96 |
| 4.6 | The feed-forward loop of [1] | 98 |
| 4.7 | Overview of the learning framework | 109 |
| 4.8 | Example of a parent graph | 114 |
| 4.9 | Results of learning the Triangle Tireworld domain and the Elevators domain . . | 116 |

List of Tables

| | | |
|------|--|----|
| 3.1 | Execution of LF1T in inferring $\pi(N_1)$ of Example 3.1 | 28 |
| 3.2 | LF1T algorithm with least generalization, running on the example of figure 2.1. | 29 |
| 3.3 | Execution of LF1T with ground resolution in inferring $\pi(N_1)$ of Example 3.2 . | 32 |
| 3.4 | Learning time of LF1T for Boolean networks up to 15 nodes | 33 |
| 3.5 | Learning details of LF1T on different partial state transitions sets of T-helper benchmark | 35 |
| 3.6 | Wolfram's Rule 110 | 37 |
| 3.7 | LF1T algorithm with Bias I on Rule 110 in Torus world | 39 |
| 3.8 | LF1T algorithm with Biases I and II on Rule 110 in Torus world | 40 |
| 3.9 | Memory use and learning time of LF1T for Boolean networks up to 15 nodes with the alphabetical variable ordering | 53 |
| 3.10 | Experimental results of 1000 runs of LF1T with random variable orderings . . | 53 |
| 3.11 | Execution of LF1T with least specialization on step transitions of figure 2.1 . . | 62 |
| 3.12 | Memory use and learning time of LF1T for Boolean networks benchmarks up to 23 nodes in the same condition as in [2] | 66 |
| 3.13 | Results of 1000 runs of LF1T with least specialization for Boolean networks benchmarks: random transition orderings | 66 |
| 3.14 | Properties of the different LF1T algorithms | 66 |
| 4.1 | Execution of LFkT on traces of figure 4.4 | 77 |
| 4.2 | Execution of LFkT on some traces of the figure 4.4 | 94 |

Chapter 1

Introduction

This thesis studies methods for the automatic construction of model of the dynamics of a system from the observation of its state transitions. Learning system dynamics from the observation of state transitions has many applications in multi-agent systems, robotics and bioinformatics alike. Knowing its environment dynamics can allow an agent to identify the consequences of its actions more precisely. This Knowledge can be used by agents and robots for planning and scheduling. In bioinformatics, learning the dynamics of biological systems can correspond to the identification of the influence of genes and can help the understanding of their interactions.

Figure 1.1 gives the big picture of our work. Given some raw data on the process, like time series data of gene expression, we assume a discretization of those data in the form of state transitions. From those state transitions, according to the semantic of the system dynamics, we propose different inference algorithms that model the system as a logic program. The semantic of system dynamics can differ regarding the synchronism of its variables, the determinism of its

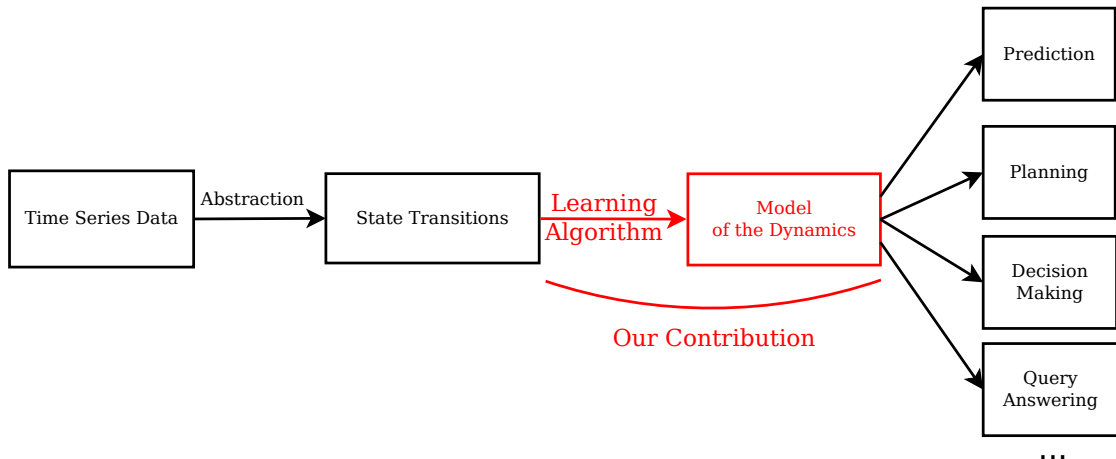


FIGURE 1.1: Big picture of our contribution: assuming a discretization of time series data of a system as state transitions executions, we propose a method to automatically model the system dynamics.

changes and the influence of its history. In a synchronous system all transition rules are applied at the same time: all variable values can change during a state transition. In a deterministic system there is only one possible transition for each state of the system. An asynchronous semantic implies that at most one transition rule can be applied in a transition, it means that only one variable can change its value at a time. Asynchronism usually leads to non-deterministic behaviors: from the same state there are multiple possible transitions. When system dynamics are memoryless, the next state of the system only depends on the current state. More complex systems can have memory, the value of its components can influence its behavior over multiple time steps. It can have the form of delayed influences: a variable could influence the state of the system k times step later. Memory can also represent duration or accumulation: a variable could need to keep a certain value or pass by a sequence of values to have a certain effect on the state of the system. In this thesis, we propose several modelings and learning algorithms to tackle those different semantics. Our methodology is based on inductive logic programming, a discipline that mixes machine learning and logic programming techniques.

Machine learning is a discipline concerned with the development, analysis and implementation of automated methods that allow a machine to evolve through a learning process, and perform tasks that are difficult or impossible to fill by more conventional algorithmic means. The purpose of Machine Learning algorithms is to provide to a computer-controlled system the capability to adapt its behavior through the analysis of empirical data from a database or sensors. The difficulty lies in the fact that the set of all possible behaviors given all possible entries quickly becomes too complex to describe using conventional programming languages. In machine learning, we entrust programs to fit a model to avoid this complexity and to use it operationally. In addition, this model is adaptive: it has to take into account the evolution of the information for which the response behavior has been validated, the so-called learning. This allows self-improvement of the system analysis, which is one of the possible forms of artificial intelligence. These programs, according to their degree of development, possibly incorporate probabilistic data processing capabilities, data analysis from sensors, recognition (voice recognition, pattern recognition, writing, etc.), data-mining, theoretical computer science, etc.

Logic programming is a form of programming, where a program is defined by a set of elementary facts and logic rules associating these facts with their (more or less) direct consequences. These facts and rules are operated by a theorem prover or inference engine according to a question or request. This approach is much more flexible than the definition of a sequence of instructions that the computer would perform. Logic programming is usually considered a declarative programming approach rather than imperative. A logic program focuses more on the “what” than the “how”, it declares a problem, not how to solve it. In logic programming, it is the inference engine that takes care of the “how”, it does most of the computation, where the developer as well as the user care about what they want. This programming paradigm is particularly suited to the needs of artificial intelligence, which it is one of the main tools. Logic programming

plays on the ambivalence between declarative and procedural representation. However, logic programs always keep a pure logical interpretation to ensure their correction and, because of their declarative nature, are more abstract than their imperative equivalent while remaining executable.

Inductive Logic Programming (ILP) is an approach to machine learning which uses logic programming techniques. From a base of facts and expected results, divided into positive and negative examples, an ILP system tries to deduce a logic program which confirms the positive and refutes negative examples. We can summarize this principle in the following way: positive examples + negative examples + knowledge base = rules. ILP is defined as the intersection of inductive machine learning and logic programming. Like in machine learning, the goal of ILP is the development of methods that construct hypotheses from observations to extract knowledge from experience. In contrast to most other approaches to inductive learning, inductive logic programming is interested in properties of inference rules, in convergence of algorithms, and in the computational complexity of procedures.

In some previous ILP works, state transitions systems are represented with logic programs, in which the dynamics that rule the environment changes are represented by a logic rules. Based on this idea, the method we propose learn logic programs from state transitions. In this method the system dynamics are represented by a logic program that is a set of transitions rules. The learning setting can be summarize as follows. We are given a set of state transitions and the goal is to induce a logic program that realizes the given transition relations. More generally speaking, our goal is to construct a model of the observed system, a model that can explain those observations.

Let's now consider a simple example in Figure 1.2. We are considering a group of friends who are having a chat discussion about going or not to a party. The raw data of the system in this example are the traces of their discussion, like the one of Figure 1.3.

The discretization into state transitions is made as follows. Here Pink invites Blue, Brown and Yellow to a party, the system can be represented by three Boolean variables which represent if each character joins or not the group. Here we do not need to represent Pink by a variable since it will always have the same value because she joins her own party of course. The current



FIGURE 1.2: The four characters of our system: Blue, Brown and Yellow invited by Pink.

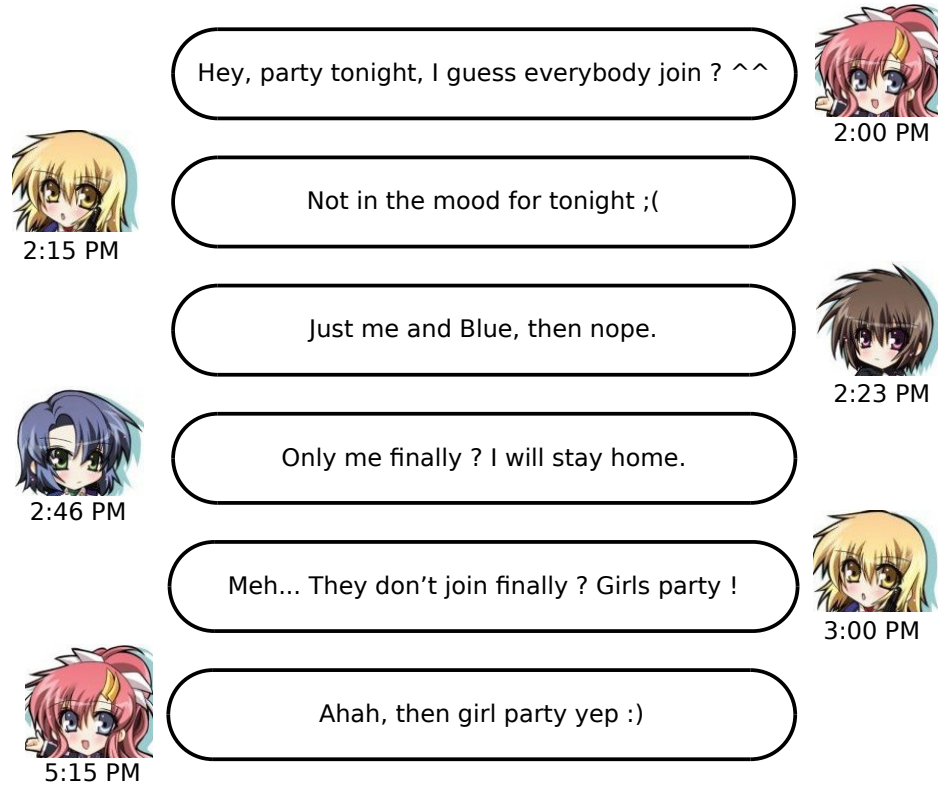









FIGURE 1.3: Trace of the chat discussion of the characters.



FIGURE 1.4: Discretization of the chat discussion into state transitions.

state of the system is the group of people who join the party. Each character can see all the messages and according to these messages the group evolves, i.e. someone joins or leaves. In this example, Pink wants to know why her friends decide to join or not the group. What we want to model is the relationship between the different characters: who likes whom and who dislike whom.

Figure 1.4 shows the discretization of the chat discussion into state transitions. At the beginning Pink thinks that everybody will join the party, so that the initial state of the system is . Then at 2:15pm, Yellow says that she does not want to join so that the state changes to . Eight minutes later, Brown decides to not join the group and the state becomes . Blue is now alone in the group and this situation does not fit his preferences because he decides to not join the party and the group becomes empty. It seems that Pink will be alone tonight, but at 3:00pm Yellow changes her mind and decides to not let Pink alone, the state of the system becomes . Finally, only Yellow joins the party of Pink. From those state transitions the method we propose can learn a logic program that represents the influences among the character. This logic program is a set of transition rules that can realize the observed evolution

of the group. Those transitions rules could be equivalent to the following informal statements:
 joins only if  joins;  joins only if  and  joins;  joins only if  does not join.

Knowing those relationships, Pink can now understand the reaction of her friends and can deduce why only Yellow joins her party. In this example, the variables are Boolean and the semantics is synchronous, deterministic and memoryless. But we could imagine a more dynamics complex with more variable values. We could have a representation that used three value variables: the character said he will join, he will not join or he has not decided yet. Also memory could be considered: the choice of a character could also depend on his own previous choice. For example, we can have a character that does not change his mind once he has said he will join or not. Or we could have another character that always waits for the decision of some others to decide what to do. Non-determinism is also possible in this kind of example: a character can make different choices in the same condition according to his mood, mood that is unknown, leading to non-deterministic behavior in the decision of this character. In this thesis we propose several modeling and learning algorithm to address those different types of dynamics.

1.1 Background

In this section we recall the background of three main fields of research which our contribution belongs to, that are: Machine Learning in section 1.1.1, Logic Programming in section 2.1 and Inductive Logic Programming in section 1.1.3. This chapter is build upon different sources that mainly come from wikipedia and review papers.

1.1.1 Machine Learning

Machine learning is used to provide to computer systems or machines the perception of their environment. For example, a machine learning system can allow a robot to learn how to walk, the robot initially knowing nothing about the coordination of movements for walking. The robot could start by making random movements, and then, selecting and focusing on movements allowing it to move forward, will gradually establish a more efficient way of walking. Another example is handwriting recognition. Handwriting recognition is a complex task because the occurrences of the same character are never exactly equal. A machine learning system can be designed to learn how to recognize characters by observing examples of the known characters. Handwriting recognition algorithms are used everyday to sort out our letter according to the addresses.

In 1959, Arthur Samuel [3] defined machine learning as a "field of study that gives computers the ability to learn without being explicitly programmed". Tom M. Mitchell [4] provided a widely quoted, more formal definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E ". This definition is notable for its defining machine learning in fundamentally operational rather than cognitive terms. The famous question "Can machines think?" of Alan Turing's proposal [5], is simplified to "Can machines do what we (as thinking entities) can do?" [6].

The learning algorithms can be categorized according to the learning style they use:

1.1.1.1 Supervised learning

If the classes are predetermined and examples are known, the system learns to classify according to a classification model; this is called supervised learning (or discriminant analysis). An expert (or oracle) must first labels the examples. The process occurs in two phases. During the first phase (offline, called learning or training), it comes to determine a model of the tagged data.

The second phase (online, called test) is to predict the label of a new data, knowing the previously learned model. Sometimes it is better not to associate a single class, but a probability of belonging to each of the predetermined classes (this is called probabilistic supervised learning).

The linear discriminant analysis and support vector machine (SVM) are typical examples. Another example is the detection of common symptoms with other known patients (examples): the system can categorize new patients given their medical analyzes and estimate the risk (probability) of developing a particular disease.

Supervised learning assumes that all examples are complete and labeled. The term semi-supervised learning is used when data (or "tags") are missing. Performed probabilistically or not, it aims to reveal the underlying distribution of examples in their description space. The model must deal with unlabeled examples that can provide some information. In medicine, it can be an aide to diagnosis or choice of the least expensive diagnostic tests.

We use the terms partially supervised learning (probability or not), when the labeling data is partial. This is the case when a model states that a given data do not belong to a class A, but perhaps to a class B or C. For example, A, B and C could be three diseases mentioned in the context of a differential diagnosis.

1.1.1.2 Unsupervised learning

When the system or operator has only examples, without labels, and the number of classes and their nature were not predetermined, it is called unsupervised learning or clustering. Here, no expert is required, the algorithm must discover by himself more or less hidden data structure. Data partitioning is an unsupervised learning algorithm. Here, the system must - in the feature space (the sum of the data) - target data according to their available attributes to classify homogeneous group of examples. Similarity is generally calculated using a distance function between pairs of examples. It is then up to the operator to combine or infer meaning for each group and clusters patterns or groups of groups in their space. Various mathematical tools and software can help. It is also called data regression analysis (model fit by least squares procedure or other optimization of a cost function). If the approach is probabilistic (that is to say, each example, instead of being classified as a single class, is characterized by a set of probabilities of belonging to each class), it is called soft clustering (as opposed to hard clustering).

For an epidemiologist who would want to try to make emerge explanatory hypotheses from a broad set of liver cancer victims, the computer could differentiate different groups, the epidemiologist then seek to associate with various explanatory factors, geographical origin, genetics, habits and consumption practices, exposures to various potentially or actually toxic agents (heavy metals, toxins, etc.).

1.1.1.3 Reinforcement Learning

The algorithm learns behavior as an observation. The action of the algorithm on the environment produces a return value that guides the algorithm learning. According to [7], reinforcement learning is learning what to do - how to map situations to actions - so as to maximize a numerical reward. In this paradigm, the learner is not informed of what actions it should take. It must find, according to the situation, which actions are the most rewarding by trying them out. In complex problem, actions can have cascading effect: not only affect the immediate reward, but also the future situations and thus, all subsequent rewards. This exploration–exploitation dilemma is the characteristic features of reinforcement learning.

One of the domain of application of reinforcement learning is robotics. In robotics, one of the major challenges is the ability to adapt to change: the capacity that allows to perform new tasks which were previously unknown. Reinforcement learning enables a robot to autonomously discover an optimal behavior through trial-and-error interactions with its environment. Instead of explicitly detailing the solution to a problem, in reinforcement learning the designer of a control task provides feedback in terms of a scalar objective function that measures the one-step performance of the robot.

1.1.2 Logic Programming

The story of logic programming starts in the early 1930s with Jacques Herbrand. In [8], Herbrand lays a first stone by setting the conditions for the validity of an automatic demonstration. Later, in 1953, Quine [9] gave an original rule of inference, defined for Zeroth-order logic. In that time, it was of little interest except to improve the calculation of the logic circuits. In 1958, McCarthy [10] was already proposing to use logic as a declarative language for knowledge representation, considering a theorem prover as a problem solver. The idea was to divide problem solving between the knowledge engineer, responsible for the validity of the application expressed logically, and the inference engine, responsible for a valid and effective implementation. Then, in 1965, Robinson [11] proposed his resolution method: he based automatic demonstration on the conditions of Herbrand, with a *reductio ad absurdum* using logical statements with clausal form and a resolution rule, providing an extension of the Quine's rule to first-order logic. The firsts approaches showed that the idea was there, but an efficient expression was still remaining to be found.

The first logic programming applications (1964-1969) were questions/answers systems. Absys (1969) was probably the first programming language based assertions [12, 13]. The notion of Logic programming comes from the debates of that time about knowledge representation in artificial intelligence. In Stanford and Edinburgh, McCarthy and Kowalski, were for a declarative representation and in MIT, Minsky and Papert, were for a procedural representation.

However, it is at the MIT that emerged Planner (1969) [14], a programming language based on logic. A part of Planner, Micro-Planner [15], was used by Winograd for SHRDLU [16], a program for the computer understanding of English. Planner invoked procedural plans based on goals and assertions, and used backtracking to spare the little quantity of available memory of these times. From Planner, drifted QA-4 [17], Conniver [18] (1972), Popler [19] (1973), QLISP [20] (1976) and Ether [21] (1981).

However, Hayes and Kowalski in Edinburgh were trying to reconcile declarative approach and knowledge representation with the Planner procedural approach. In 1972, Hayes developed an equation language Golux [22], who could invoke various procedures by altering the operation of the inference engine. Kowalski also showed that the SL-resolution addressed the implications as reducing procedures goals. In 1971, Colmerauer and Kowalski saw that the clausal form could represent a formal grammars and inference engine could be used for the analysis of texts, some engines providing a bottom-up analysis, and Kowalski's SL-resolution providing a top-down analysis. The next year, they developed the procedural interpretation of the implications, and established that clauses could be restricted to Horn clauses, corresponding to implications where conditions and consequences are atomic statements. Then, in 1973, Colmerauer and Roussel proposed the Prolog language [23] as a tool to describe a world in French, and then

deal with the question about this world, Prolog being used for both analysis and synthesis in French as well as for the reasoning producing the answers. This first Prolog diffused quickly. The interest of Prolog for natural language querying databases led to a configurator for Solar computers whose drift different query systems in English, French, Portuguese and German.

In 1976, the first port of Prolog for microcomputer was made. Finally, in 1977, Warren developed a Prolog compiler in Edinburgh, who provided the performances that Prolog was lacked. And this Prolog became the standard we know nowadays.

1.1.2.1 Probabilistic Logic Programming

Probabilistic logic, also called probability logic or probabilistic reasoning, consist in the integration of probability theory into deductive logic structure. The result is a richer and more expressive formalism with a broad range of possible application in areas like artificial intelligence, argumentation theory, bioinformatics, game theory, statistics. Probabilistic logics attempt to find a natural extension of traditional logic truth tables: the results they define are derived through probabilistic expressions instead.

The term "probabilistic logic" was first used in a paper by Nils Nilsson published in 1986 [24], where the truth values of sentences are probabilities. The proposed semantical generalization induces a probabilistic logical entailment, which reduces to ordinary logical entailment when the probabilities of all sentences are either 0 or 1. This generalization applies to any logical system for which the consistency of a finite set of sentences can be established.

A difficulty with probabilistic logics is that they tend to multiply the computational complexities of their probabilistic and logical components. Other difficulties include the possibility of counter-intuitive results, such as those of Dempster-Shafer theory [25]. The need to deal with a broad variety of contexts and issues has led to many different proposals. Those different approaches can be categorized into two different main classes: those logics that attempt to make a probabilistic extension to logical entailment [26], and those that attempt to address the problems of uncertainty and lack of evidence.

The central concept in the theory of subjective logic [27] are opinions about some of the propositional variables involved in the given logical sentences. A binomial opinion applies to a single proposition and is represented as a 3-dimensional extension of a single probability value to express various degrees of ignorance about the truth of the proposition. For the computation of derived opinions based on a structure of argument opinions, the theory proposes respective operators for various logical connectives, such as multiplication (AND), comultiplication (OR), division (UN-AND) and co-division (UN-OR) of opinions [28] as well as conditional deduction (MP) and abduction (MT) [29].

Approximate reasoning formalism proposed by fuzzy logic [30] can be used to obtain a logic in which the models are the probability distributions and the theories are the lower envelopes [31]. In such a logic the question of the consistency of the available information is strictly related with the one of the consistency of partial probabilistic assignment.

Markov logic networks [32] implement a form of uncertain inference [33] based on the maximum entropy principle [34, 35] the idea that probabilities should be assigned in such a way as to maximize entropy, in analogy with the way that Markov chains assign probabilities to finite state machine transitions.

Systems such as Pei Wang's Non-Axiomatic Reasoning System [36] or Ben Goertzel's Probabilistic Logic Networks [37] add an explicit confidence ranking, as well as a probability to atoms and sentences. The rules of deduction and induction incorporate this uncertainty, thus sidestepping difficulties in purely Bayesian approaches to logic (including Markov logic), while also avoiding the paradoxes of Dempster-Shafer theory. The implementation of PLN attempts to use and generalize algorithms from logic programming, subject to these extensions.

The theory of evidential reasoning [38] also defines non-additive probabilities of probability (or epistemic probabilities) as a general notion for both logical entailment (provability) and probability. The idea is to augment standard propositional logic by considering an epistemic operator K that represents the state of knowledge that a rational agent has about the world. Probabilities are then defined over the resulting epistemic universe K_p of all propositional sentences p , and it is argued that this is the best information available to an analyst. From this view, Dempster-Shafer theory appears to be a generalized form of probabilistic reasoning.

1.1.3 Inductive Logic Programming

Inductive Logic Programming is an approach to machine learning which uses logic programming techniques. From a base of facts and expected results, divided into positive and negative examples, an ILP system tries to deduce a logic program which confirms the positive and infirm negative examples. We can summarize this principle in the following way: positive examples + negative examples + knowledge base = rules. In [39], ILP is defined as the intersection of inductive machine learning and logic programming. Like in machine learning, the goal of ILP is the development of methods that construct hypotheses from observations to extract knowledge from experience. In contrast to most other approaches to inductive learning, inductive logic programming is interested in properties of inference rules, in convergence of algorithms, and in the computational complexity of procedures [40].

Many classical machine learning techniques, like the Top-Down-Induction-of-Decision-Tree family [41], require a limited knowledge representation such as propositional logic. In many cases, this limitation can be a problem because knowledge can only be expressed in a first-order logic, or one of its variant. Inductive logic programming avoid this problem by using computational logic as the representational mechanism for hypotheses and observations.

From computational logic, inductive logic programming inherits its representational formalism, its semantical orientation, and various well-established techniques. Many inductive logic programming systems benefit from using the results of computational logic. Additional benefit could potentially be derived from making use of work on termination, types and modes, knowledge-base updating, algorithmic debugging, abduction, constraint logic programming, program synthesis, and program analysis.

Inductive logic programming extends the theory and practice of computational logic by investigating induction rather than deduction as the basic mode of inference. Whereas present computational logic theory describes deductive inference from logic formulas provided by the user, inductive logic programming theory describes the inductive inference of logic programs from instances and background knowledge. In this manner, ILP may contribute to the practice of logic programming, by providing tools that assist logic programmers to develop and verify programs.

Chapter 2

Preliminaries

In this chapter we introduce the preliminaries notions needed for the understanding of our contribution. Section [2.1](#) provides formal definitions of several notions about logic programs. Notions that we will use and extend later in the contributions chapters to formalize our approaches. Section [2.2](#) provides formalizations of Boolean networks, that is the main model formalism we use as benchmarks in our experiments. Section [2.3](#) provides a formalization of the dynamics of state transitions systems as logic programs. Thus, it provides the formal intuition behind the inferences methods used in our framework and also show how the output of our algorithms can be used to realize state transitions. Finally, section [2.4](#) gives a summary of the contribution of this thesis. This summary is also a kind of PhD report: it describe the evolution of the work chronologically to provide to the reader the intuition of the motivations behind each extension of the framework.

2.1 Logic Programming

We now recall some preliminaries of logic programming. We consider a propositional language L that is built from a finite set of propositional constants p, q, r, \dots and the logical connectives \neg, \wedge and \leftarrow .

Definition 2.1 (Atom). An atom is of the form $p(t_1, \dots, t_n)$, where p is a predicate symbol and each t_i is a term. If each t_i is ground, then the atom is said to be ground.

Definition 2.2 (Literal). A literal is either an atom or an atom preceded by the symbol " \neg ". The former is referred to as a positive literal, while the latter is referred to as a negative literal. Let l be a literal, the complement of l is denoted \bar{l} . The complement of a positive literal l is its negation $\neg l$ and the complement of a negative literal $\neg l'$ is its atom l' .

Definition 2.3 (Herbrand Universe). The Herbrand Universe of a program P , denoted HU_P , is the set of all terms which can be formed with the functions and constants in P .

A ground *normal logic program* (NLP) is a set of *rules* of the form

$$p \leftarrow p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n \quad (2.1)$$

where p and p_i 's are atoms ($n \geq m \geq 1$). For any rule R of the form (2.1), the atom p 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 (2.1) as $b(R) = \{p_1, \dots, p_m, \neg p_{m+1}, \dots, \neg p_n\}$, and the atoms appearing in the body of R positively and negatively as $b^+(R) = \{p_1, \dots, p_m\}$ and $b^-(R) = \{p_{m+1}, \dots, p_n\}$, respectively. A rule R of the form (1) is interpreted as follows: $h(R)$ is true if all elements of $b^+(R)$ are true and none of the elements of $b^-(R)$ is true. When $b^+(R) = b^-(R) = \emptyset$, the rule is called a *fact rule*. The rule (1) is a *Horn clause* iff $m=n$.

Definition 2.4 (Herbrand Base). The Herbrand Base of a program P , denoted by \mathcal{B} , is the set of all atoms in the language of P .

Definition 2.5 (Interpretation). Let \mathcal{B} be the Herbrand Base of a logic program P . An *interpretation* is a subset of \mathcal{B} . If an interpretation is the empty set, it is denoted by ϵ .

Definition 2.6 (Model). An interpretation I is a model of a program P if $b^+(R) \subseteq I$ and $b^-(R) \cap I = \emptyset$ imply $h(R) \in I$ for every rule R in P .

For a logic program P and an interpretation I , the *immediate consequence operator* (or T_P operator) [42] is the mapping $T_P : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$:

$$T_P(I) = \{ h(R) \mid R \in P, b^+(R) \subseteq I, b^-(R) \cap I = \emptyset \}. \quad (2.2)$$

The state transitions of a logic program P can be represented as a set of pairs of interpretations $(I, T_P(I))$.

Definition 2.7 (Consistency). Let R be a rule and (I, J) be a state transition. R is *consistent* with (I, J) iff $b^+(R) \subseteq I$ and $b^-(R) \cap I = \emptyset$ imply $h(R) \in J$. 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 .

Definition 2.8 (Subsumption). Let R_1 and R_2 be two rules. If $h(R_1) = h(R_2)$ and $b(R_1) \subseteq b(R_2)$ then R_1 *subsumes* R_2 . Let P be a logic program and R be a rule. If there exists a rule $R' \in P$ that *subsumes* R then P *subsumes* R .

We say that a rule R_1 is *more general* than another rule R_2 if R_1 subsumes R_2 .

Example 2.1. Let R_1 and R_2 be the two following rules: $R_1 = (a \leftarrow b)$, $R_2 = (a \leftarrow a \wedge b)$, R_1 subsumes R_2 because $(b(R_1) = \{b\}) \subset (b(R_2) = \{a, b\})$. When R_1 appears in a logic program P , R_2 is useless for P , because whenever R_2 can be applied, R_1 can be applied.

In ILP, search mainly relies on generalization and specialization that are dual notions. Generalization is usually considered as an induction operation, and specialization as a deduction operation. In [43], the author define the minimality and maximality of the generalization and specialization operations as follows.

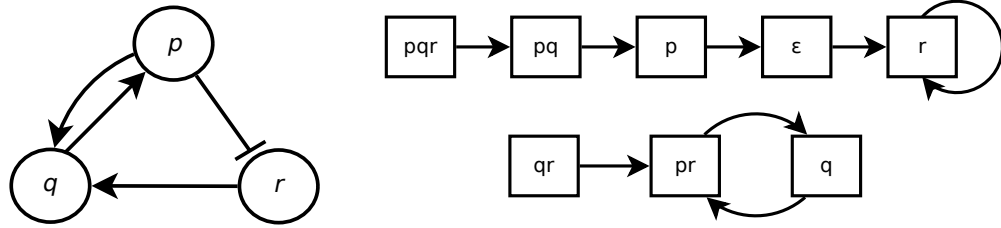
Definition 2.9 (Generalization operator [43]). A generalization operator maps a conjunction of clauses S onto a set of minimal generalizations of S . A **minimal generalization** G of S is a generalization of S such that S is not a generalization of G , and there is no generalization G' of S such that G is a generalization of G' .

Example 2.2. $C_1 = (p \wedge q)$ is a minimal generalization of $C_2 = (p \wedge q \wedge r)$. But $C_3 = (p)$ is not a minimal generalization of C_2 . Because C_3 is a generalization of C_1 and C_1 is a generalization of C_2 .

Definition 2.10 (Specialization operator [43]). A specialization operator maps a conjunction of clauses G onto a set of maximal specializations of G . A **maximal specialization** S of G is a specialization of G such that G is not a specialization of S , and there is no specialization S' of G such that S is a specialization of S' .

The body of a rule of the form (1) can be considered as a clause, so that, the Definition 2.9 and 2.10 can also be used to compare the body of two rules.

Example 2.3. $C_1 = (p \wedge q \wedge r)$ is a maximal specialization of $C_2 = (p \wedge q)$. But C_1 is not a maximal specialization of $C_3 = (p)$. Because C_2 is a specialization of C_3 and C_1 is a specialization of C_2 .

FIGURE 2.1: A Boolean network B_1 (left) and its state transition diagram (right)

2.2 Boolean network

A Boolean network is a simple discrete representation widely used in bioinformatics [44–46]. A *Boolean network* [44] is a pair (N, F) with $N = \{n_1, \dots, n_k\}$, a finite set of nodes (or variables), and $F = \{f_1, \dots, f_k\}$, a corresponding set of Boolean functions $f_i : \mathbb{B}^n \rightarrow \mathbb{B}$, with $\mathbb{B} = \{0, 1\}$. $n_i(t)$ represents the value of n_i at time step t , and equals either 1 (expressed) or 0 (not expressed). A vector (or *state*) $s(t) = (n_1(t), \dots, n_k(t))$ is the expression of the nodes of N at time step t . There are 2^k possible distinct states for each time step. The state of a node n_i at the next time step $t + 1$ is determined by $n_i(t + 1) = f_i(n_{i_1}(t), \dots, n_{i_p}(t))$, with n_{i_1}, \dots, n_{i_p} the nodes directly influencing n_i , called *regulation nodes* of n_i . A Boolean network can be represented by its *interaction graph* (see Figure 2.1 left), but its precise regulation relations can only be represented by the Boolean functions (see Example 2.4). From any Boolean network, we can compute the *state transition diagram* (see Figure 2.1 right) which represents the transitions between $n_i(t)$ and $n_i(t + 1)$. In the case of a gene regulatory network, nodes represent genes and Boolean functions represent their relations.

Example 2.4. Figure 2.1 shows the interaction graph and the state transitions diagram of a Boolean network B_1 composed of the three following variables: $\{a, b, c\}$. The Boolean functions of B_1 are f_a, f_b, f_c , which are respectively the following Boolean functions of a, b and c :

$$f_a = \neg a \wedge (b \vee c), \quad f_b = a \wedge c, \quad f_c = \neg a$$

Let us consider that the Boolean network B_1 , whose graph is depicted in Figure 2.1, is a gene regulatory network so that a, b, c are genes. According to the interaction graph of B_1 : a is not only an *activator* of b and an *inhibitor* of c but also its own inhibitor. The gene b is an activator of a and the gene c is activator of both a and b . According to the Boolean functions of B_1 in Example 2.4, to activate a , either b or c has to be present but if a is present, it will prevent its own expression at the next step (f_a). The activation of b requires both a and c to be expressed at the same time step; if one of them is not expressed at time step t then b will not be expressed at $t + 1$ (f_b). The presence of a is enough to prevent the expression of c , so that if a is expressed at time step t then c will not be expressed at $t + 1$ (f_c).

2.3 Representing Dynamics in Logic Programs

In this section we present the logic-based representation of dynamical systems proposed in [2], which is a key issue for inductive learning of them. In ILP, a first-order representation is used for a relational concept [47], and we simply follow this line of research. In particular, we do not propose any new learning scheme for generalization and abstraction which are not directly related to dynamics. For instance, if a particle A and a particle B have the same physical properties, then a rule to decide the position of A after a perturbation is added must be the same as a rule for B with the same kind of perturbation. Then, identification of such a rule involves the dynamics, but the names A and B are not crucial so that we can generalize them to be a variable in a common rule. In this section we show two such representations to deal with dynamics: one is based on a first-order notation with the time argument, and the other does not use the time argument.

Symbolic representation of dynamic changes has been studied in knowledge representation in AI such as situation calculus [48] and event calculus [49], which are mostly suitable for virtual action sequences. In real-world applications, however, the state of the world changes concurrently from time to time, and all elements in the world may change often *synchronously*. Then, to represent discrete time directly in the simplest way, we can use the time argument in a relational representation: for each relation $p(\mathbf{x})$ among the objects, where p is a predicate and \mathbf{x} is a tuple of its arguments, we can consider its state at time t as $p(\mathbf{x}, t)$. In this way, we shall represent any atom $A = p(\mathbf{x})$ at time t by putting the time argument of the predicate as $A^t = p(\mathbf{x}, t)$. Then, a rule in a logic program of the form (2.1) can be made a dynamic rule in the first-order expression of the form:

$$A^{t+1} \leftarrow A_1^t \wedge \cdots \wedge A_m^t \wedge \neg A_{m+1}^t \wedge \cdots \wedge \neg A_n^t. \quad (2.3)$$

The rule (2.3) means that, if A_1, \dots, A_m are all true at time step t and A_{m+1}, \dots, A_n are all false at the same time step t , then A is true at the next time step $t + 1$. Note that this kind of dynamic rules is first-order even if the original rule is propositional. Then, any first-order NLP that is a set of rules of the form (2.3) becomes an acyclic program, in which the stable model semantics and the supported model semantics coincide. Moreover, we can simulate state transition of Boolean networks using this representation and the T_P operator [50].

Inoue [50] shows a translation of a Boolean network $N = (V, F)$ into a logic program $\tau(N)$ such that $\tau(N)$ is a set of rules of the form (2.3): For each $v_i \in V$, convert its Boolean function $f_i(v_{i_1}(t), \dots, v_{i_k}(t))$ into a DNF formula¹ $\bigvee_{j=1}^{l_i} B_{i,j}^t$, where $B_{i,j}$ is a conjunction of literals, then generate l_i rules with v_i^{t+1} as the head and $B_{i,j}^t$ as a body for each $j = 1, \dots, l_i$. Given a state $S(t) = (v_1(t), \dots, v_n(t))$ at time step t , let $J^t = \{v_i^t \mid v_i \in$

¹If no f_i is given to v_i , we assume the identity function for f_i , i.e., $v_i(t+1) = v_i(t)$.

V , $v_i(t)$ is true in $S(t)$. Then the translation τ has the property that the trajectory of N from an initial state $S(0) = (v_1(0), \dots, v_n(0))$ can be precisely simulated by the sequence of interpretations, $J^0, J^1, \dots, J^k, J^{k+1}, \dots$, where $J^{k+1} = T_{\tau(N)}(J^k) \cap \{v_i^{t+1} \mid v_i \in V\}$ for $k \geq 0$ [50].

Example 2.5. Consider the Boolean network $N_1 = (V_1, F_1)$, where $V_1 = \{p, q, r\}$, and F_1 and the corresponding NLP $\tau(N_1)$ are as follows.

$$\begin{array}{ll} F_1 : & p(t+1) = q(t), \\ & q(t+1) = p(t) \wedge r(t), \\ & r(t+1) = \neg p(t). \end{array} \quad \begin{array}{ll} \tau(N_1) : & p(t+1) \leftarrow q(t), \\ & q(t+1) \leftarrow p(t) \wedge r(t), \\ & r(t+1) \leftarrow \neg p(t). \end{array}$$

The state transition diagram for N_1 is depicted in Figure 2.1.²

Starting from the interpretation $J^0 = \{q(0), r(0)\}$, which means that q and r are true at time 0, its transitions with respect to the $T_{\tau(N_1)}$ operator are given as $J_1 = \{p(1), r(1)\}$, $J_2 = \{q(2)\}$, $J_3 = \{p(3), r(3)\}$, \dots , which corresponds to the trajectory $qr \rightarrow pr \rightarrow q \rightarrow pr \rightarrow \dots$ of N_1 . Here $pr \rightarrow q \rightarrow pr$ is a cycle attractor. N_1 has also a point attractor $r \rightarrow r$ whose basin of attraction is $\{pqr, pq, p, \epsilon, r\}$.

The second way to represent dynamics of Boolean networks is based on a recent work on the semantics of logic programming. Instead of using the above direct representation (2.3), we can consider another representation without the time argument. That is, we consider a logic program as a set of rules of the form (2.1). In [50], a Boolean network N is further translated to a propositional NLP $\pi(N)$ from $\tau(N)$ by deleting the time argument from every literal A^t appearing in $\tau(N)$. Then, we can simulate the trajectory of N from any state $S(0)$ also by the orbit of the interpretation $I^0 = \{v_i \in V \mid v_i(0) \text{ is true}\}$ with respect to the $T_{\pi(N)}$ operator, i.e., $I^{t+1} = T_{\pi(N)}(I^t)$ for $t \geq 0$. Moreover, we can characterize the attractors of N based on the *supported class semantics* [51] for $\pi(N)$.

A *supported class* of a logic program P [51] is a non-empty set \mathcal{S} of Herbrand interpretations satisfying:

$$\mathcal{S} = \{T_P(I) \mid I \in \mathcal{S}\}. \quad (2.4)$$

Note that I is a supported model of P iff $\{I\}$ is a supported class of P . A supported class \mathcal{S} of P is *strict* if no proper subset of \mathcal{S} is a supported class of P . Alternatively, \mathcal{S} is a strict supported class of P iff there is a directed cycle $I_1 \rightarrow I_2, \dots \rightarrow I_k \rightarrow I_1$ ($k \geq 1$) in the state transition diagram induced by T_P such that $\{I_1, I_2, \dots, I_k\} = \mathcal{S}$ [51]. A strict supported class of $\pi(N)$ thus exactly characterizes an attractor of a Boolean network N .

²Each interpretation is concisely represented as a sequence of atoms instead of a set of atoms in examples, e.g., pq means $\{p, q\}$ and the empty string ϵ means \emptyset .

Example 2.6. Consider the Boolean network N_1 in Example 2.5 again. The NLP

$$\begin{aligned}\pi(N_1) : \quad p &\leftarrow q, \\ q &\leftarrow p \wedge r, \\ r &\leftarrow \neg p,\end{aligned}$$

is obtained from the first-order NLP $\tau(N_1)$ in Example 2.5 by removing the time argument from each literal. Notice that this logic program is not acyclic, since $\pi(N_1)$ has both positive and negative feedback loops: The positive loop appears between p and q , while the negative one exists in the dependency cycle to r through p . In this case, behavior of a corresponding Boolean network is not obvious.³

The state transition diagram induced by the $T_{\pi(N_1)}$ operator is the same as the diagram in Figure 2.1. The orbit of pqr with respect to $T_{\pi(N_1)}$ becomes $pqr, pq, p, \epsilon, r, r, \dots$ and the orbit of qr is qr, pr, q, pr, \dots . We here verify that there are two supported classes of $\pi(N_1)$, $\{\{r\}\}$ and $\{\{p, r\}, \{q\}\}$, which respectively correspond to the point attractor and the cycle attractor of N_1 .

In the following, we can use a logic program either with the time argument in the form of (2.3) or without the time argument in the usual form (2.1) for learning. To simplify the discussion, however, we will mainly use NLPs without the time argument in basic algorithms.

³The reason why behavior becomes complex in the existence of feedbacks is biologically justified as follows. Each positive loop in a Boolean network is related to reinforcement and existence of multiple attractors, while each negative loop is the source of periodic oscillations involved in homeostasis [52].

2.4 Contribution

Learning complex networks becomes more and more important. Learning system dynamics has many applications in multi-agent systems, robotics and bioinformatics alike. Knowledge of system dynamics can be used by agents and robots for planning and scheduling. In bioinformatics, learning the dynamics of biological systems can correspond to the identification of the influence of genes and can help the understanding of their interactions. In this thesis we tackle the induction problem of such dynamical systems in terms of NLP learning from state transitions. Our contribution is a framework composed of several algorithms for learning from interpretation transitions.

2.4.1 First Steps

In [2], we firstly tackled the induction problem of such dynamical systems in terms of NLP learning from synchronous state transitions. Given any state transition diagram we proposed an algorithm, **LF1T**, that can learn an NLP that exactly captures the system dynamics. Learning is performed only from positive examples, and produces NLPs that consist only of rules to make literals true. The iterative character of **LF1T** has applications in bioinformatics, cellular automata, multi-agent systems and robotics. We can imagine an agent or a robot that learns the dynamics of its environment from its observations. Knowing its environment dynamics can allow an agent to identify the consequences of its actions more precisely. **LF1T** can also be used to learn these consequences according to the state of the world step-by-step. Aggregating more and more observations, the agent becomes able to predict the evolution of the world more precisely and can use this knowledge for planning and scheduling.

2.4.2 Getting Stronger

The performances of the first implementations were not sufficient to tackle model with more than 15 variables. Lot of improvement could be done regarding the representation of the rules in the implementation at this time. Improving the performance was our next move. In [53], we proposed a new version of the **LF1T** algorithm based on Binary Decision Diagrams (BDDs) [54, 55]. A BDD is a canonical representation of a Boolean formula which has been successfully used in many research fields such as Boolean satisfiability solvers [56], data mining [57], ILP [58] and abduction [59, 60]. The main concern of this version is the memory usage of the algorithm and its run time efficiency. In previous algorithms, **LF1T** uses resolution techniques to generalize rules and reduce the size of the output NLP. The novelty of our approach is the adaptation of these techniques to the BDD structure. Here, we develop a method to perform **LF1T** operations on a BDD that also realizes usual BDD merging operations as well as novel

simplification operations. We represent an NLP by a set of BDD structures where each BDD encodes rules with the same head literal. Assuming that rules respect a variable ordering, our data structure is similar to an Ordered BDD (OBDD) [61, 62]. In our approach, each BDD represents a formula in disjunctive normal form that defines whether a literal is true at the next time step. Because **LFIT** does not learn negative rules, our structure only represents rules that imply the head literal to be true. In that sense it can also be considered as a Zero-suppressed Binary Decision Diagram (ZDD) [63]. Using a BDD representation we can also merge the common part of rules and learn the same NLP with less memory usage than in previous versions of **LFIT**. One weak point of the previous **LFIT** algorithm is that learning becomes slower and slower as the NLP learned becomes bigger because it has to check more and more rules. In practice, the compact representation of the BDD structure reduces the sensitivity of the **LFIT** learning time to the NLP size. This version allows to learn Boolean networks up to 23 variables (8,388,608 state transitions) in about one hour on an Intel Core I7 (3610QM, 2.3GHz) with 4GB of RAM. This algorithm is presented in section 3.2.

2.4.3 The Quest of Minimality

After that, we focused on the minimality of the rules and the NLPs learned by **LFIT**. Our goal was to learn all minimal conditions that imply a variable to be true in the next state, e.g. all prime implicant conditions. In bioinformatics, for a gene regulatory network, it corresponds to all minimal conditions for a gene to be activated/inhibited. It can be easier and faster to perform model checking on Boolean networks represented by a compact NLP than the set of all state transitions. Knowing the minimal conditions required to perform the desired state transitions, a robot can optimize its actions to achieve its goals with less energy consumption. From a technical point of view, for the sake of memory usage and reasoning time, a small NLP could also be preferred in multi-agent and robotics applications. For this purpose, in [64], we proposed a new version of the **LFIT** algorithm based on specialization. Specialization is usually considered the dual of generalization in ILP [43, 65, 66]. Many incremental learning systems use both generalization and specialization to revise hypothesis according to new observations/examples. Where generalization occurs when a hypothesis does not explain a positive example, specialization is used to refine a hypothesis that implies a negative example.

In [67], prime implicants are defined for DNF formula as follows: a clause C , implicant of a formula ϕ , is prime if and only if none of its proper subset $S \subset C$ is an implicant of ϕ . In this work, explanatory induction is considered, while in our approach prime implicants are defined in the **LFIT** framework. Knowing the Boolean functions, prime implicants could be computed by Tison's consensus method [68] and its variants [69]. The novelty of our approach, is that we compute prime implicants incrementally during the learning of the Boolean function. In its previous version, **LFIT** uses resolution techniques to generalize rules and reduces the size of

the output NLP. This technique generates hypotheses by *generalization* from the most specific clauses until every positive transitions are covered. Compared to previous versions, the novelty of our new approach is that, now, we generate hypotheses by *specialization* from the most general clauses until no negative transition is covered. The main weak point of the previous **LFIT** algorithms is that the output NLPs depends on variable/transition ordering. Our new method guarantees that the NLPs learned contain only minimal conditions for a variable to be true in the next state. This algorithm is presented in section 3.3.

2.4.4 Facing Delays

With rule minimality being guaranteed, we started to consider how our framework can contribute to model Biological system. 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 [70] or the DNA damage repair [71]. 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 [72] 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 aim was to give an efficient and valuable approach to learn such dynamics.

We extended our framework by designing an algorithm that takes multiple sequences of state transition as input. This algorithm builds a normal logic program that captures the delayed dynamics of a system. While the previous algorithm dealt only with 1-step transitions (i.e., we assume the state of the system at time t depends only of its state at time $t - 1$), in [73], we proposed an approach that is able to consider k -step transitions (sequence of at most k state transitions). This means that we are able to capture delayed influences in the inductive logic programming methodology. This algorithm is presented in section 4.1.

2.4.5 Uncertain Future

Our last contributions was to learn non-deterministic systems (multiple next states from the same state). We extended the *LFIT* framework to learn probabilistic dynamics by proposing an extension of *LFIT* for learning from uncertain state transitions. We developed an algorithm that learns a set of deterministic logic programs, each state transition observed is realized by at least one of them. Considering that the system learned is purely non-deterministic, given a state I , all possible next states J is given by those programs. Then, we can provide the likelihood of each rule by simply counting how many transitions they realize correctly.

This algorithm has been used by David Martinez during his internship at the Inoue Laboratory. Our algorithm was used as to learn the model of the environment where a robot evolved. It inferred rules from observation of the consequence of a robot actions. Then a planner used those rules to decide what next action the robot should perform to achieve his goal.

Chapter 3

Learning From Interpretation Transitions

In this chapter we introduce the basis of our methods by providing modelisation and algorithm to learn Boolean systems with synchronous deterministic semantics. First, in section 3.1 we introduce our first contribution to the LFIT framework that is a method based on resolution to learn logic program from state transitions. Then we provide an efficient data structure based on Binary Decision Diagram and an adapted algorithm in section 3.2. Finally we provide a method based on specialization that guarantee to output minimal rules in section 3.3.

The algorithms of the first section have been published in the *Machine Learning Journal* [2]. The BDD optimization methods we present in section 3.2 have been published in *The 24th International Conference on Inductive Logic Programming (ILP 2013)* [53]. And the algorithms of section 3.3 have been published the following year in ILP 2014 [64].

3.1 Learning from 1-step Transitions

Learning complex networks becomes more and more important. Learning system dynamics has many applications in multi-agent systems, robotics and bioinformatics alike. Knowledge of system dynamics can be used by agents and robots for planning and scheduling. In bioinformatics, learning the dynamics of biological systems can correspond to the identification of the influence of genes and can help the understanding of their interactions. In [2], we firstly tackled the induction problem of such dynamical systems in terms of NLP learning from synchronous 1-step state transitions.

LFIT is an *anytime algorithm*, that is, whenever we process a set E of state transitions, we will guarantee that the result of learning is a logic program P which completely represents the dynamics of the transitions E so that a dynamical system is represented by P . For example, from the state transitions of figure 2.1, we can learn a logic program equivalent to the Boolean network they belong to.

For learning, we assume that the Herbrand base \mathcal{B} is finite. The state transitions of E can be seen as (positive) examples/observations of transition of the system. 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 **LFIT**, the Herbrand base \mathcal{B} is assumed to be finite. In the input E , a state transitions is represented by a pair of Herbrand interpretations. The output of **LFIT** is an NLP that realizes all state transitions. To construct an NLP for **LFIT** we can use a bottom-up method, which generates hypotheses by *generalization* from the most specific clauses to explain positive examples that have not been covered yet. The pseudo-code of **LFIT** is given as follows.

LFIT(E : pairs of Herbrand interpretations, P : an NLP)

1. If $E = \emptyset$ then output P and stop;
2. Pick $(I, J) \in E$, and put $E := E \setminus \{(I, J)\}$;
3. For each $A \in J$, let

$$R_A^I := \left(A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j \right); \quad (3.1)$$

4. If R_A^I is not subsumed by any rule in P , then $P := P \cup \{R_A^I\}$ and simplify P by generalizing some rules in P and removing all clauses subsumed by them;
5. Return to 1.

The **LF1T** algorithm can be used with or without an initial NLP P_0 . When only the examples E are given as input, **LF1T** is initially called by $\mathbf{LF1T}(E, \emptyset)$. If an initial NLP P_0 is given, $\mathbf{LF1T}(E, P_0)$ is executed. **LF1T** first constructs the most specific rule R_A^I for each positive literal A appearing in $J = T_P(I)$ for each $(I, J) \in E$. We do not construct any rule to make a literal false. The rule R_A^I is then possibly generalized when another transition from E makes A true, which is computed by several generalization methods. The first generalization method we consider is based on *resolution*. The resolution principle by Robinson [74] is well known as a deductive method, but its naïve use can be applied to a generalization method. In the following, for a literal l , \bar{l} denotes the *complement* of l , i.e., when A is an atom, $\bar{A} = \neg A$ and $\overline{\neg A} = A$. We firstly consider a resolution between two ground rules as follows.

Definition 3.1 (Naïve/Ground Resolution). Let R_1, R_2 be two rules and l be a literal such that $h(R_1) = h(R_2)$, $l \in b(R_1)$ and $\bar{l} \in b(R_2)$. If $(b(R_2) \setminus \{\bar{l}\}) \subseteq (b(R_1) \setminus \{l\})$ then the *ground resolution* of R_1 and R_2 (upon l) is defined as

$$res(R_1, R_2) = \left(h(R_1) \leftarrow \bigwedge_{L_i \in b(R_1) \setminus \{l\}} L_i \right). \quad (3.2)$$

In particular, if $(b(R_2) \setminus \{\bar{l}\}) = (b(R_1) \setminus \{l\})$ then the *ground resolution* is called the *naïve resolution* of R_1 and R_2 (upon l). In this particular case, the rules R_1 and R_2 are said to be *complementary* to each other with respect to l .

3.1.1 Generalization by Naïve Resolution

When naïve resolution is used, we need an auxiliary set P_{old} of rules to globally store subsumed rules, which increases monotonically. P_{old} is set to \emptyset at first. When a generated rule is newly added at Step 4 in the pseudo-code of **LF1T**, we try to find a rule $R' \in P \cup P_{old}$ such that (a) $h(R') = h(R)$ and (b) $b(R)$ and $b(R')$ differ in the sign of only one literal l . If there is no such rule R' , then R is just added to P ; otherwise, add R and R' to P_{old} then add $res(R, R')$ to P in a recursive call of Step 4.

Example 3.1. Consider the state transitions in Fig. 2.1. By giving the state transitions step by step, the NLP $\pi(N_1) = \{\#11, \#14, \#19\}$ is obtained in Table 3.1, where $\#n$ is the rule ID.

We now examine the correctness of the **LF1T** algorithm in terms of its completeness and soundness. A program P is said to be *complete* for a set E of pairs of interpretations if $J = T_P(I)$ holds for any $(I, J) \in E$. On the other hand, P is *sound* for E if for any $(I, J) \in E$ and any $J' \in 2^B$ such that $J' \neq J$, $J' \neq T_P(I)$ holds. A deterministic learning algorithm is *complete* (resp. *sound*) for E if its output program is complete (resp. sound) for E . We use the following subsumption relation between programs: Given two logic programs P_1 and P_2 , P_1 *theory-subsumes* P_2 if for any rule $R \in P_2$, there is a rule $R' \in P_1$ such that R' subsumes R .

Algorithm 1 LF1T(E, P)

```

1: INPUT: a set  $E$  of pairs of Herbrand interpretations and an NLP  $P$ 
2: OUTPUT: an NLP  $P$ 

3:  $P_{old}$ : NLP
4:  $P_{old} \leftarrow \emptyset$ 
5: while  $E \neq \emptyset$  do
6:   Pick  $(I, J) \in E$ ;  $E := E \setminus \{(I, J)\}$ 
7:   for each  $A \in J$  do
8:      $R_A^I := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (B \setminus I)} \neg C_j$ 
9:     AddRule( $R_A^I, P, P_{old}$ )
10:  end for
11: end while
return  $P$ 

```

Algorithm 2 AddRule(R, P, P_{old}) (with naïve resolution)

```

1: INPUT : a rule  $R$  and two NLPs  $P$  and  $P_{old}$ 

2: if  $R$  is subsumed by a rule of  $P$  then
3:    $P_{old} := P_{old} \cup \{R\}$  return
4: end if
5: for each rule  $R_P$  of  $P$  subsumed by  $R$  do
6:    $P := P \setminus \{R_P\}$ 
7:    $P_{old} := P_{old} \cup \{R_P\}$ 
8: end for

9:  $P := P \cup \{R\}$ 
10: // Check for generalizations
11: for each rule  $R'$  of  $P \cup P_{old}$  with  $h(R) = h(R')$  do
12:   for each  $l \in b(R)$  such that  $\bar{l} \in b(R')$  do
13:     if  $b(R) \setminus \{l\} = b(R') \setminus \{\bar{l}\}$  then
14:        $P_{old} := P_{old} \cup \{R\}$ 
15:        $R^{lg} := h(R) \leftarrow \bigwedge_{L_i \in b(R) \setminus \{l\}} L_i$ 
16:       AddRule( $R^{lg}, P, P_{old}$ )
17:     end if
18:   end for
19: end for

```

Theorem 3.2 (Completeness of LF1T with naïve resolution). *Given a set E of pairs of interpretations, LF1T with naïve resolution is complete for E .*

Proof. For any pair of interpretations $(I, J) \in E$, it is verified that the rule R_A^I determines the value of A in the next state of I correctly for any $A \in J$. On the other hand, for any atom $A \notin J$, the value of A in the next state of I becomes false by R_A^I and the T_P operator. Hence, the set of rules $P^* = \{R_A^I \mid (I, J) \in E, A \in J\}$ is complete for the transitions in E . Since a rule R derived by the naïve resolution of R_1 and R_2 subsumes R_1 and R_2 by Definition 3.1,

TABLE 3.1: Execution of **LFIT** in inferring $\pi(N_1)$ of Example 3.1

| Step | $I \rightarrow J$ | Operation | Rule | ID | P | P_{old} |
|------|--------------------------|----------------|---|----|---------------|-------------|
| 1 | $qr \rightarrow pr$ | R_p^{qr} | $p \leftarrow \neg p \wedge q \wedge r$ | 1 | 1 | \emptyset |
| | | R_r^{qr} | $r \leftarrow \neg p \wedge q \wedge r$ | 2 | 1,2 | |
| 2 | $pr \rightarrow q$ | R_q^{pr} | $q \leftarrow p \wedge \neg q \wedge r$ | 3 | 1,2,3 | |
| 3 | $q \rightarrow pr$ | R_p^q | $p \leftarrow \neg p \wedge q \wedge \neg r$ | 4 | | |
| | | $res(4, 1)$ | $p \leftarrow \neg p \wedge q$ | 5 | 2,3,5 | + 1,4 |
| | | R_r^q | $r \leftarrow \neg p \wedge q \wedge \neg r$ | 6 | | |
| | | $res(6, 2)$ | $r \leftarrow \neg p \wedge q$ | 7 | 3,5,7 | + 2,6 |
| 4 | $pqr \rightarrow pq$ | R_p^{pqr} | $p \leftarrow p \wedge q \wedge r$ | 8 | | |
| | | $res(8, 1)$ | $p \leftarrow q \wedge r$ | 9 | 3,5,7,9 | + 8 |
| | | R_q^{pqr} | $q \leftarrow p \wedge q \wedge r$ | 10 | | |
| | | $res(10, 3)$ | $q \leftarrow p \wedge r$ | 11 | 5,7,9,11 | + 3,10 |
| 5 | $pq \rightarrow p$ | R_p^{pq} | $p \leftarrow p \wedge q \wedge \neg r$ | 12 | | |
| | | $res(12, 4)$ | $p \leftarrow q \wedge \neg r$ | 13 | 5,7,9,11,13 | + 12 |
| | | $res(13, 9)$ | $p \leftarrow q$ | 14 | 7,11,14 | + 5,9,13 |
| 6 | $p \rightarrow \epsilon$ | | | | | |
| 7 | $\epsilon \rightarrow r$ | R_r^ϵ | $r \leftarrow \neg p \wedge \neg q \wedge \neg r$ | 15 | | |
| | | $res(15, 6)$ | $r \leftarrow \neg p \wedge \neg r$ | 16 | 7,11,14,16 | + 15 |
| 8 | $r \rightarrow r$ | R_r^r | $r \leftarrow \neg p \wedge \neg q \wedge r$ | 17 | | |
| | | $res(17, 15)$ | $r \leftarrow \neg p \wedge \neg q$ | 18 | 7,11,14,16,18 | + 17 |
| | | $res(18, 7)$ | $r \leftarrow \neg p$ | 19 | 11,14,19 | + 7,16,18 |

$P' = (P^* \setminus \{R_1, R_2\}) \cup \{R\}$ theory-subsumes P^* . Then, P' is also complete for E , since $T_{P'}$ and T_P agree with their transitions. Since the (theory-)subsumption relation is transitive, an output program P , which is obtained by repeatedly applying naïve resolutions, theory-subsumes P^* . Hence, P is complete for E .

□

The implication of Theorem 3.2 is very important: *For any set of 1-step state transitions, we can construct an NLP that captures the dynamics in the transitions.* In other words, there is no (deterministic) state transitions diagram that cannot be expressed in an NLP. It is also important to guarantee the soundness of the learning algorithm, that is, it never overgeneralizes any state transitions rule. The soundness can be obtained from the completeness when the transition from any interpretation is deterministic like the assumption in this paper (that is why it is stated as a corollary), but we show a more precise proof for it.

Corollary 3.3 (Soundness of **LFIT** with naïve resolution). *Given a set E of pairs of interpretations, **LFIT** with naïve resolution is sound for E .*

Proof. It is easy to see that the program P^* in the proof of Theorem 3.2 satisfies the soundness. Any naïve resolution $R = res(R_1, R_2)$ for any $R_1, R_2 \in P^*$ deletes only one literal l such that $l \in b(R_1)$ and $\bar{l} \in R_2$. Assume that $R_1 = R_A^{I_1}$ and $R_2 = R_A^{J_1}$ for some $(I_1, J_1) \in E$ and

| Step | $I \rightarrow J$ | Operation | Rule | ID | P | P_{old} |
|--------------|-------------------------------------|----------------|---|-------|------------------|-------------|
| 1 | $\epsilon \rightarrow pqr$ | R_p^ϵ | $p \leftarrow \neg p \wedge \neg q \wedge \neg r$ | 1 | 1 | \emptyset |
| | | R_q^ϵ | $q \leftarrow \neg p \wedge \neg q \wedge \neg r$ | 2 | 1,2 | |
| | | R_r^ϵ | $r \leftarrow \neg p \wedge \neg q \wedge \neg r$ | 3 | 1,2,3 | |
| 2 | $pqr \rightarrow p$ | R_p^{pqr} | $p \leftarrow p \wedge q \wedge r$ | 4 | 1,2,3,4 | |
| 3 | $p \rightarrow pq$ | R_p^p | $p \leftarrow p \wedge \neg q \wedge \neg r$ | 5 | 2,3,4,6 | +1,5 |
| | | $lg(5, 1)$ | $p \leftarrow \neg q \wedge \neg r$ | 6 | | |
| | | q_q^p | $q \leftarrow p \wedge \neg q \wedge \neg r$ | 7 | 3,4,6,8 | +2,7 |
| $lg(7, 2)$ | $q \leftarrow \neg q \wedge \neg r$ | 8 | | | | |
| 4 | $pq \rightarrow pq$ | R_p^{pq} | $p \leftarrow p \wedge q \wedge \neg r$ | 9 | 3,6,8,10 | +4,9 |
| | | $lg(9, 4)$ | $p \leftarrow p \wedge q$ | 10 | | |
| | | q_q^{pq} | $q \leftarrow p \wedge q \wedge \neg r$ | 11 | 3,6,8,10,12 | +7,11 |
| $lg(11, 7)$ | $q \leftarrow p \wedge \neg r$ | 12 | | | | |
| 5 | $q \rightarrow qr$ | R_q^q | $q \leftarrow \neg p \wedge q \wedge \neg r$ | 13 | 3,6,10,15 | +2,13 |
| | | $lg(13, 2)$ | $q \leftarrow \neg p \wedge \neg r$ | 14 | | +8,12,14 |
| | | $lg(14, 12)$ | $q \leftarrow \neg r$ | 15 | | |
| 6 | $qr \rightarrow r$ | R_r^q | $r \leftarrow \neg p \wedge q \wedge \neg r$ | 16 | 6,10,15,17 | +3,16 |
| | | $lg(16, 3)$ | $r \leftarrow \neg p \wedge \neg r$ | 17 | | |
| | | R_r^{qr} | $r \leftarrow \neg p \wedge q \wedge r$ | 18 | 6,10,15,17,19 | +18 |
| $lg(18, 16)$ | $r \leftarrow \neg p \wedge q$ | 19 | | | | |
| 7 | $r \rightarrow pr$ | R_p^r | $p \leftarrow \neg p \wedge \neg q \wedge r$ | 20 | 6,10,15,17,19,21 | +20 |
| | | $lg(20, 1)$ | $p \leftarrow \neg p \wedge \neg q$ | 21 | | +22 |
| | | R_r^r | $r \leftarrow \neg p \wedge \neg q \wedge r$ | 22 | 6,10,15,21,24 | |
| $lg(22, 3)$ | $r \leftarrow \neg p \wedge \neg q$ | 23 | | | | |
| 8 | $pr \rightarrow p$ | $lg(23, 19)$ | $r \leftarrow \neg p$ | 24 | | |
| | | R_p^{pr} | $p \leftarrow p \wedge \neg q \wedge r$ | 25 | 15,24,27 | +25 |
| | | $lg(25, 5)$ | $p \leftarrow p \wedge \neg q$ | 26 | | +10,26 |
| $lg(26, 10)$ | $p \leftarrow p$ | 27 | 15,24,27,28 | +6,21 | | |
| $lg(26, 21)$ | $p \leftarrow \neg q$ | 28 | | | | |

TABLE 3.2: **LF1T** algorithm with least generalization, running on the example of figure 2.1.

$(I_2, J_2) \in E$. Then, $b(R)$ is satisfied by any partial interpretation I' such that $I' = I_1 \cap I_2 = I_1 \setminus \{l\} = I_2 \setminus \{\bar{l}\}$. Considering total interpretations that are extensions of I' , there are only two possibilities, i.e., I_1 and I_2 . Since $A = h(R)$ belongs to both $J_1 = T_{P^*}(I_1)$ and $J_2 = T_{P^*}(I_2)$, it also belongs to $T_{P'}(I_1)$ and $T_{P'}(I_2)$, where $P' = (P^* \setminus \{R_1, R_2\}) \cup \{R\}$. Applying the same argument to all atoms in any $J = T_{P^*}(I)$ for any interpretation I , we have $J = T_{P'}(I)$. This arguments can be further applied to all naïve resolutions, so that $T_P(I)$ is the same as $T_{P^*}(I)$ for the final NLP P .

□

3.1.2 Generalization by Ground Resolution

Using naïve resolution, $P \cup P_{old}$ possibly contains all patterns of rules constructed from the Herbrand base \mathcal{B} in their bodies. In our second implementation of **LF1T**, ground resolution is

used as an alternative generalization method in **AddRule**. This replacement of resolution leads to a lot of computational gains, since we do not need P_{old} any more: Every generalization which can be found in P_{old} can be found in P by ground resolution.

Proposition 3.4. *All generalized rules obtained from $P \cup P_{old}$ by naïve resolution can be obtained using ground resolution on P .*

Proof. Let $R_1 \in P$ and $R_2 \in P_{old}$ be ground complementary rules with respect to a literal $l \in b(R_1)$. Then, $h(R_1) = h(R_2)$, $\bar{l} \in b(R_2)$ and $(b(R_1) \setminus \{l\}) = (b(R_2) \setminus \{\bar{l}\})$ hold. Suppose that by naïve resolution, $R_3 = res(R_1, R_2)$ is put into P and that R_1 is put into P_{old} in **AddRule**. By $R_2 \in P_{old}$, there has been a rule R_4 in P such that R_4 subsumes R_2 , that is, $b(R_4) \subseteq b(R_2)$. We can also assume that $\bar{l} \in b(R_4)$ because otherwise l has been resolved upon by the naïve resolution between R_2 and some rule in P and thus R_1 must have been put into P_{old} . Then, the rule $R_5 = res(R_1, R_4)$ is obtained by ground resolution, and $b(R_5) = (b(R_1) \setminus \{l\}) = (b(R_2) \setminus \{\bar{l}\})$. Hence R_5 is equivalent to R_3 . □

Ground resolution can be used in place of naïve resolution to learn an NLP from traces of state transition. In this case, we can simplify Algorithm 1 by deleting Lines 3 and 4 and by replacing Line 9 with **AddRule**(R_A^I , P). Algorithm 26 describes the new **AddRule** which adds and simplifies rules using ground resolution.

As in the case of naïve resolution, we can prove the correctness, i.e., the completeness and soundness of **LF1T** with ground resolution.

Theorem 3.5 (Completeness of **LF1T** with ground resolution). *Given a set E of pairs of interpretations, **LF1T** with ground resolution is complete for E .*

Proof. As in the proof of Theorem 3.2, if a program P is complete for E , a program P' that theory-subsumes P is also complete for E . By Proposition 3.4, any rule produced by naïve resolution can be generated by ground resolution. Then, if P and P' are respectively obtained by naïve resolution and ground resolution, P' theory-subsumes P . Since P is complete for E by Theorem 3.2, P' is complete for E . □

Corollary 3.6 (Soundness of **LF1T** with ground resolution). *Given a set E of pairs of interpretations, **LF1T** with ground resolution is sound for E .*

Proof. By Theorem 3.5, a program P output by **LF1T** with ground resolution is complete for E . Then, as in the proof of Corollary 3.3, P is shown to be sound for E .

Algorithm 3 AddRule(R, P) (with ground resolution)

```

1: INPUT : a rule  $R$  and a NLP  $P$ 

2: for each rule  $R_P$  of  $P$  do
3:   if  $R$  is subsumed by  $R_P$  then return
4:   end if
5:   if  $R$  subsumes  $R_P$  then
6:      $P := P \setminus \{R_P\}$ 
7:   else
8:     // Check for generalizations
9:     if  $h(R) = h(R_P)$  then
10:      if  $\exists l \in b(R)$  such that  $\bar{l} \in b(R_P)$  then
11:        if  $b(R) \setminus \{l\}$  is subsumed by  $b(R_P) \setminus \{\bar{l}\}$  then
12:           $R^r := h(R) \leftarrow \bigwedge_{L_i \in b(R) \setminus \{l\}} L_i$ 
13:          AddRule( $R^r, P$ ) return
14:        end if
15:        if  $b(R) \setminus \{l\}$  subsumes  $b(R_P) \setminus \{\bar{l}\}$  then
16:           $R_P^r := h(R_P) \leftarrow \bigwedge_{L_i \in b(R_P) \setminus \{\bar{l}\}} L_i$ 
17:          AddRule( $R_P^r, P$ )
18:          AddRule( $R, P$ ) return
19:        end if
20:      end if
21:    end if
22:  end if
23: end for
24:  $P := P \cup \{R\}$ 

```

□

Example 3.2. Consider again the state transitions in Fig. 2.1. Using ground resolution, the NLP $\pi(N_1) = \{\#11, \#14, \#19\}$ is obtained in Table 3.3.

Comparing Examples 3.1 and 3.2, all rules generated by naïve resolution are obtained by ground resolution too. By avoiding the use of P_{old} , however, we can reduce time and space for learning. As the next theorem shows, ground resolution has much complexity gain compared with naïve resolution, when learning is done with the input of complete 1-step state transitions from all 2^n interpretations, where n is the size of the Herbrand base \mathcal{B} . In the propositional case, n is the number of propositional atoms, which correspond to the number of nodes in a Boolean network. We here assume that each operation of subsumption and resolution can be performed in time $O(1)$ by assuming a bit-vector data structure.

Theorem 3.7. Using naïve version, the memory use of the **LF1T** algorithm is bounded by $O(n \cdot 3^n)$, and the time complexity of learning is bounded by $O(n^2 \cdot 9^n)$, where $n = |\mathcal{B}|$. On the other hand, with ground resolution, the memory use is bounded by $O(2^n)$, which is the maximum size of P , and the time complexity is bounded by $O(4^n)$.

TABLE 3.3: Execution of **LF1T** with ground resolution in inferring $\pi(N_1)$ of Example 3.2

| Step | $I \rightarrow J$ | Operation | Rule | ID | P |
|------|--------------------------|----------------|---|----|------------|
| 1 | $qr \rightarrow pr$ | R_p^{qr} | $p \leftarrow \neg p \wedge q \wedge r$ | 1 | 1 |
| | | R_r^{qr} | $r \leftarrow \neg p \wedge q \wedge r$ | 2 | 1,2 |
| 2 | $pr \rightarrow q$ | R_q^{pr} | $q \leftarrow p \wedge \neg q \wedge r$ | 3 | 1,2,3 |
| 3 | $q \rightarrow pr$ | R_p^q | $p \leftarrow \neg p \wedge q \wedge \neg r$ | 4 | |
| | | $res(4, 1)$ | $p \leftarrow \neg p \wedge q$ | 5 | 2,3,5 |
| | | R_r^q | $r \leftarrow \neg p \wedge q \wedge \neg r$ | 6 | |
| | | $res(6, 2)$ | $r \leftarrow \neg p \wedge q$ | 7 | 3,5,7 |
| 4 | $pqr \rightarrow pq$ | R_p^{pqr} | $p \leftarrow p \wedge q \wedge r$ | 8 | |
| | | $res(8, 5)$ | $p \leftarrow q \wedge r$ | 9 | 3,5,7,9 |
| | | R_q^{pqr} | $q \leftarrow p \wedge q \wedge r$ | 10 | |
| | | $res(10, 3)$ | $q \leftarrow p \wedge r$ | 11 | 5,7,9,11 |
| 5 | $pq \rightarrow p$ | R_p^{pq} | $p \leftarrow p \wedge q \wedge \neg r$ | 12 | |
| | | $res(12, 5)$ | $p \leftarrow q \wedge \neg r$ | 13 | |
| | | $res(13, 9)$ | $p \leftarrow q$ | 14 | 7,11,14 |
| 6 | $p \rightarrow \epsilon$ | | | | |
| 7 | $\epsilon \rightarrow r$ | R_r^ϵ | $r \leftarrow \neg p \wedge \neg q \wedge \neg r$ | 15 | |
| | | $res(15, 7)$ | $r \leftarrow \neg p \wedge \neg r$ | 16 | 7,11,14,16 |
| 8 | $r \rightarrow r$ | R_r^r | $r \leftarrow \neg p \wedge \neg q \wedge r$ | 17 | |
| | | $res(17, 7)$ | $r \leftarrow \neg p \wedge r$ | 18 | |
| | | $res(18, 16)$ | $r \leftarrow \neg p$ | 19 | 11,14,19 |

Proof. In both P and P_{old} , the maximum size of the body of a rule is n . There are n possible heads and 3^n possible bodies for each rule: Each element of \mathcal{B} can be either positive, negative or absent in the body of a rule. This means that both $|P|$ and $|P_{old}|$ are bounded by the size in $O(n \cdot 3^n)$. The memory use in the algorithm is thus $O(n \cdot 3^n)$. In practice, however, $|P|$ is less than or equal to $O(2^n)$ for the following reason. In the worst case, P contains only rules of size n ; if P contains a rule with m literals ($m < n$), this rule subsumes 2^{n-m} rules which cannot appear in P . That is why we can consider only two possibilities for each literal, i.e., positive and negative occurrences of the literal (and no blank) to estimate the size $|P|$. Furthermore, P does not contain any pair of complementary rules, so that the complexity is further divided by n , that is, $|P|$ is bounded by $O(n \cdot 2^n / n) = O(2^n)$. But $|P_{old}|$ remains in the same complexity and the memory use of the algorithm in practice is still $O(n \cdot 3^n)$.

When adding a rule to P in **AddRule** using naïve resolution, we have to compare it with all rules in $P \cup P_{old}$, then this operation has a complexity of $O(n \cdot 3^n)$. Hence, using naïve resolution, the complexity of **LF1T** is $O(\sum_{k=1}^{n \cdot 3^n} k)$, where k represent the number of rules in $P \cup P_{old}$, which increases during the process until it finally belongs to $O(n \cdot 3^n)$. Therefore, the complexity of learning with naïve version is $O(\sum_{k=1}^{n \cdot 3^n} k)$, which is then equal to $O(n^2 \cdot 3^{2n-1}) = O(n^2 \cdot 9^n)$. On the other hand, using ground resolution, the memory use of the **LF1T** algorithm is $O(2^n)$, which is the maximum size of P . The complexity of learning is then $O(\sum_{k=1}^{2^n} k)$, which is equal to $O((2^n(2^n + 1))/2) = O(2^{2n-1}) = O(4^n)$.

□

By Theorem 3.7, given the set E of complete state transitions, which has the size $O(2^n)$, the complexity of $\mathbf{LFIT}(E, \emptyset)$ with ground resolution is bounded by $O(|E|^2)$. On the other hand, the worst-case complexity of learning with naïve resolution is $O(n^2 \cdot |E|^{4.5})$.

3.1.3 Experiments

In this section, we evaluate our learning methods through experiments. We apply our LFIT algorithms to learn Boolean networks [75] in Section 3.1.3.1, and apply LFIT to identification of cellular automata [76] in Section 3.1.3.3.

3.1.3.1 Learning Boolean Networks

We here run our learning programs on some benchmarks of Boolean networks taken from Dubrova and Teslenko [77], which include those networks for control of flower morphogenesis in *Arabidopsis thaliana* [78], budding yeast cell cycle regulation [79], fission yeast cell cycle regulation [80] and mammalian cell cycle regulation [81]. However, since our problem setting for learning is different from that for computing attractors in [77], we needed to reproduce these inverse problems, which are made as follows. Firstly, we construct an NLP $\tau(N)$ from the Boolean function of a Boolean network N using the translation in Section 2.3, where each Boolean function is transformed to a DNF formula. Then, we get all possible 1-step state transitions of N from all 2^B possible initial states I^0 's by computing all stable models of $\tau(N) \cup I^0$ using the answer set solver `clasp`.¹ Finally, we use this set of state transitions to learn an NLP using our LFIT algorithms. Because a run of \mathbf{LFIT} returns an NLP which can contain redundant rules, the original NLP P_{org} and the output NLP P_{LFIT} can be different, but remain equivalent with respect to state transitions, that is, $T_{P_{org}}$ and $T_{P_{LFIT}}$ are identical functions.

| Name | # nodes | # rules (org./LFIT) | Naïve (CPU/learned rules) | Ground (CPU/produced rules) |
|-----------------------------|---------|---------------------|---------------------------|-----------------------------|
| <i>Arabidopsis thaliana</i> | 15 | 28 / 241 | T.O. | 13.825s / 1,446,548 |
| Budding yeast | 12 | 54 / 54 | 361s / 415,252 | 0.820s / 129,557 |
| Fission yeast | 10 | 23 / 24 | 5.208s / 45,122 | 0.068s / 16,776 |
| Mammalian cell | 10 | 22 / 22 | 5.756s / 47,222 | 0.076s / 18,955 |

TABLE 3.4: Learning time of \mathbf{LFIT} for Boolean networks up to 15 nodes

Table 3.4 shows the time of a single \mathbf{LFIT} run in learning a Boolean network for each problem in [77] on a processor Intel Core I7 (3610QM, 2.3GHz). The time limit is set to 1 hour for each experiment. We can see the good effect of using ground resolution in place of naïve resolution. The number of learned rules in each setting is also shown in Table 3.4, and is compared with

¹<http://potassco.sourceforge.net/>

the original literatures that present networks. Except *Arabidopsis thaliana*, LFIT succeeds to reconstruct the same gene regulation rules as in [77] in the first run of LFIT. However, in *Arabidopsis thaliana*, only 12 original rules are reproduced and the 16 other original rules are replaced with other learned 229 rules in the output of the first run of LFIT. Although those output rules are all minimal with respect to subsumption among them, some are subsumed by original rules. This is because, our resolution strategy is to perform resolution only when it produces a generalized rule, so other kinds of resolution, as general resolution, are not allowed. For example, from $R_1 = (p \leftarrow p \wedge q)$ and $R_2 = (p \leftarrow \neg q \wedge r)$, $R = (p \leftarrow p \wedge r)$ cannot be obtained in LFIT, since R subsumes neither R_1 nor R_2 . Then, we applied boostings twice for *Arabidopsis thaliana*, and obtained 76 rules in the first boosting, then got exactly the same 28 original rules in the second boosting. In constructing regulation rules of fission yeast, only one rule is additionally produced: $R_{15} = (x_5 \leftarrow \neg x_2 \wedge \neg x_4 \wedge x_5 \wedge x_6)$. Note that all output rules including R_{15} are minimal with respect to subsumption among them, and in particular every output rule is not subsumed by any of the original rule. This rule does not disappear with a boosting and the number of learned rules does not decrease from 24. Rules like R_{15} are not necessary to capture the whole transitions, but may give an alternative way to implement the dynamics. Hence, the same transition system can be realized in different ways. If this is considered as a redundancy, it might be useful for robustness of biological systems.

In this experiment, the algorithm needs to analyze 2^n steps of transitions to learn an NLP, where n is the number of nodes in a Boolean network. That is why our implemented programs cannot handle networks with more than 20 nodes in the benchmark; computing all 1-step transitions takes too much time, since the grounding in the answer set solver cannot handle it. In other words, the input size with more than 2^{20} is too huge to be handled, so that we cannot even start learning. Such a limitation is acceptable in the ILP literature; for example, it has been stated that networks with 10 transitions and 10 nodes are reasonably large for learning [82]. Moreover, in real biological networks, we do not observe an exponential number of the whole state transitions from all possible initial states. Hence, anytime algorithms like LFIT must be useful for such incomplete set of transitions, since learned programs are correct for any given partial set of state transitions.

3.1.3.2 Learning big benchmark from partial state transitions sets

In this experiment, the algorithm needs to analyze $2^{|B|}$ steps of transitions to learn an NLP, where n is the number of nodes in a Boolean network. That is why our implemented programs cannot handle networks with more than 20 nodes in the benchmark; since the current implementation is still too naïve to efficiently solve such big instances. Nevertheless, this limitation is acceptable in the ILP literature; for example, it has been stated that networks with 10 transitions and 10

TABLE 3.5: Learning details of **LFIT** on different partial state transitions sets of T-helper benchmark

| # of transitions | # of output rules | avg. size of output rules | # of analyzed rules | CPU time |
|------------------------|-------------------|---------------------------|---------------------|----------------|
| 5% (419,431) | 131 | 8.51 | 28,244,034 | 356.40s |
| 10% (838,861) | 116 | 8.48 | 63,643,273 (+35M) | 754s (+398s) |
| 15% (1,258,292) | 116 | 8.57 | 101,751,250 (+38M) | 1206s (+452s) |
| 20% (1,677,722) | 133 | 7.99 | 142,478,023 (+40M) | 1670s (+464s) |
| 25% (2,097,153) | 17 | 3.11 | 191,512,576 (+49M) | 2475s (+805s) |
| 30% (2,516,583) | 120 | 7.58 | 224,668,076 (+33M) | 2983s (+508s) |
| 35% (2,936,013) | 135 | 6.93 | 265,389,835 (+40M) | 4758s (+1765s) |

nodes are reasonably large for learning [82]. Moreover, in real biological networks, we do not observe an exponential number in the whole state transitions from all possible initial states.

For such big networks we can still learn interesting rules within a reasonable time from a partial set of transition.

We experiment with such partial learning using the benchmark of T-helper cell differentiation [83], which is composed of 23 nodes [77]. Here we are concerned with the evolution of the NLP learned by **LFIT** from 5% to 35% of the step transitions set. **LFIT** start with 5% of the step transitions and outputs a NLP, then it considers 5% more step transitions and outputs a new NLP, and so on until 35%. Table 3.5 presents details about the NLPs learned from these partial step transitions sets. For each increase of 5% of the input, the table shows: the number of rules of the output NLP and the average size (i.e. number of body literals) of these rules; and the last columns give the number of produced rules and CPU time. Results show that CPU time and number of produced rules slowly increase until 20% of the step transitions. Furthermore, the number of rules in the NLP and their size is almost the same. Between 20-25% we observe a cascade of generalizations which leads to a reduction of the number of rules from 122 to 17. Also, these rules are quite small compared to previous one: average size of almost 3 against 8 before. The number of produced rules and CPU time slightly increase (9 millions more rules and too time slower) compared to previous evolutions. Between 25-30%, because of the generality of the rules of the NLP, we see that the number of produced rules is quite reduced. After 30%, the evolution of the NLP is quite similar to one before 25%: the number of rules remains within the same magnitude, and the average size of rules diminishes. Finally, we observe a drastic increase of CPU time between 30% and 35% which cannot be explained only with the properties we analyzed: the number of produced rules and properties of the output NLP are similar to previous observations. The time for checking new rules depends on the size of the NLP: our hypothesis is that the size of the NLP explodes during this phase, so that checking takes more time than before. However, thanks to many simplifications, it finally leads to a NLP of 135 rules with an average size of 7.

Example 3.3. *An NLP learned from 25% of the step transitions of the benchmark T-helper (2^{20} step transitions):*

$x2 \leftarrow x1, x12, x20.$
 $x4 \leftarrow x12, x20, x3.$
 $x6 \leftarrow x12, \text{not } x19, x20, x5.$
 $x7 \leftarrow x12, x20, x6.$
 $x8 \leftarrow x12, x20.$
 $x10 \leftarrow x12, \text{not } x19, x20, x9.$
 $x13 \leftarrow x12, x20, \text{not } x23.$
 $x14 \leftarrow x12, x13, x20.$
 $x15 \leftarrow x12, x14, x20, \text{not } x8.$
 $x16 \leftarrow x12, x15, x20.$
 $x16 \leftarrow x12, x20, x4.$
 $x17 \leftarrow x12, \text{not } x16, x20.$
 $x18 \leftarrow x12, x17, x20, \text{not } x8.$
 $x19 \leftarrow x12, x18, x20.$
 $x21 \leftarrow x12, x20.$
 $x22 \leftarrow x12, x20, x21.$
 $x23 \leftarrow x12, x20, x22.$

3.1.3.3 Learning Cellular Automata

A *cellular automaton* (CA) [76] consists of a regular grid of *cells*, each of which has a finite number of possible states. The state of each cell changes synchronously in discrete time steps according to a local and identical *transition rule*. The state of a cell in the next time step is determined by its current state and the states of its surrounding cells (called *neighbors*). The collection of all cellular states in the grid at some time step is called a *configuration*. An *elementary CA* consists of a one-dimensional array of possibly infinite cells, and each cell has one of two possible states 0 (white, dead) or 1 (black, alive). A cell and its two adjacent cells form a neighbor of three cells, so there are $2^3 = 8$ possible patterns for neighbors. A transition rule describes for each pattern of a neighbor, whether the central cell will be 0 or 1 at the next time step.

In [84], Adamatzky asserts the problem to identify a CA from an arbitrary pair of its configurations. We provide a solution to this problem using LFIT. Here, we performed experiments of **LFIT** with a background theory and inductive biases to learn transition rules of cellular automata.

Example 3.4. We here pick up one of the most famous elementary CAs, known as Wolfram's Rule 110 [76], whose transition rule is given in Table 3.6. In the table, the eight possible values of the three neighboring cells are shown in the first row, and the resulting value of the central

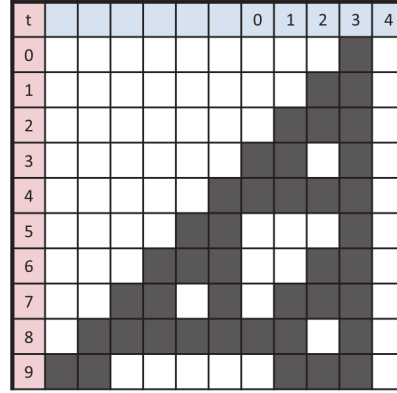


FIGURE 3.1: State changes by Wolfram's Rule 110

cell in the next time step is shown in the second row. Rule 110 is known to be Turing-complete. The pattern generated by Rule 110 from the initial configuration with only one true cell (colored black) is depicted in Fig. 3.1. In the figure, time starts at 0 and patterns are shown until time 9. The column numbers are used later, and we here assign 3 to the column with the single black cell at time 0. We see that every cell at column 4 has the state 0 through transitions, since its neighbors always have the state 100 (assuming that the invisible column 5 has the state 0 at time 0).

We here reproduce the rules for Wolfram's Rule 110 from traces of configuration changes. Although this problem is rather simple, it illustrates how the whole system of LFIT with a background theory and inductive biases works to induce NLPs for CAs.

Originally, an infinite space is assumed for the CA with Rules 110. To deal with the CA in a finite space, two approximations can be considered:

1. **Limited frame:** Observes partially some set of cells. The problem in this setting is that, from the same configuration, different transitions can occur. For example, the configurations of cells $(1, 2, 3)$ at $t = 2$ and at $t = 4$ take the same values $(1, 1, 1)$ in Fig. 3.1, but the next states are $(1, 0, 1)$ at $t = 3$ and $(0, 0, 1)$ at $t = 5$. If the frame width is only 3, then we have two mutually inconsistent transitions from the same configuration. Hence, rules are not constructed for the two edge cells but are learned only for the internal cells.
2. **Torus world:** Assumes that there is no end in the shape of circle, doughnut or sphere, which can be constructed by chaining one edge cell with the other in one-dimensional

TABLE 3.6: Wolfram's Rule 110

| Current pattern | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---------------------------|-----|-----|-----|-----|-----|-----|-----|-----|
| New state for center cell | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

| t | (4) | 1 | 2 | 3 | 4 | (1) |
|---|-----|---|---|---|---|-----|
| 0 | | | | | | |
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |

FIGURE 3.2: State changes by Wolfram’s Rule 110 in Torus world

cell patterns. Fig. 3.2 shows a torus world of size 4 and the state transitions by Rule 110 with the initial configuration $(1, 2, 3, 4) = (0, 0, 1, 0)$. The columns numbered (4) and (1) are thus identical to columns 4 and 1, respectively. Note that the configurations reach the attractor, $(1, 1, 1, 0) \rightarrow (1, 0, 1, 1) \rightarrow (1, 1, 1, 0)$.

Due to these approximations, the number of possible state transitions can be made smaller in the case of elementary CAs like Fig. 3.2. Our learning framework can handle both limited frames and torus worlds by considering adequate state transitions representation as input. For example, to represent a torus world of size 4, a configuration is represented by a vector with 6 elements $(0, 1, 2, 3, 4, 5)$: 1, 2, 3, 4 respectively represent their values in the corresponding cells, and 0, 5 respectively represent the values of cells 4 and 1 (colored gray when the value is 1). This last information can be represented in a background theory as the two rules with the time argument:

$$\begin{aligned} c(0, t) &\leftarrow c(4, t), \\ c(5, t) &\leftarrow c(1, t), \end{aligned}$$

where $c(x)$ represents a cell x and $c(x, t)$ is its state at time step t . Unfortunately, these two rules do not have corresponding rules without the time argument, since the head literals refer the time step t instead of $t + 1$. Hence, simple removal of the time argument from the both sides changes the dynamic meaning of the NLP in application of the T_P operator that infers about the next time step. Then, without the time argument, we should copy the rules for $c(4)$ to those for $c(0)$, and copy the rules for $c(1)$ to those for $c(5)$.

Now we use non-ground resolution and consider the following two biases:

- **Bias I:** The body of each rule contains at most n neighbor literals.

TABLE 3.7: LFIT algorithm with Bias I on Rule 110 in Torus world

| Step | $I \rightarrow J$ | Operation | Rule | ID | P |
|------|-----------------------------|------------------|--|----|----------------|
| 1 | 000100 \rightarrow 001100 | $R_{c(2)}^{001}$ | $c(2) \leftarrow \neg c(1) \wedge \neg c(2) \wedge c(3)$ | 1 | 1 |
| | | $R_{c(3)}^{010}$ | $c(3) \leftarrow \neg c(2) \wedge c(3) \wedge \neg c(4)$ | 2 | 1,2 |
| 2 | 001100 \rightarrow 011101 | $R_{c(1)}^{001}$ | $c(1) \leftarrow \neg c(0) \wedge \neg c(1) \wedge c(2)$ | 3 | |
| | | $lg(3, 1)$ | $c(x) \leftarrow \neg c(x-1) \wedge \neg c(x) \wedge c(x+1)$ | 4 | 2,4 |
| | | $R_{c(2)}^{011}$ | $c(2) \leftarrow \neg c(1) \wedge c(2) \wedge c(3)$ | 5 | |
| | | $res(5, 4)$ | $c(2) \leftarrow \neg c(1) \wedge c(3)$ | 6 | 2,4,6 |
| | | $R_{c(3)}^{110}$ | $c(3) \leftarrow c(2) \wedge c(3) \wedge \neg c(4)$ | 7 | |
| | | $res(7, 2)$ | $c(3) \leftarrow c(3) \wedge \neg c(4)$ | 8 | 4,6,8 |
| 3 | 011101 \rightarrow 110111 | $R_{c(1)}^{011}$ | $c(1) \leftarrow \neg c(0) \wedge c(1) \wedge c(2)$ | 9 | |
| | | $lg(9, 6)$ | $c(x) \leftarrow \neg c(x-1) \wedge c(x) \wedge c(x+1)$ | 10 | |
| | | $lg(10, 4)$ | $c(x) \leftarrow \neg c(x-1) \wedge c(x+1)$ | 11 | 8,11 |
| | | $R_{c(3)}^{110}$ | $c(3) \leftarrow c(2) \wedge c(3) \wedge \neg c(4)$ | 12 | |
| | | $R_{c(4)}^{101}$ | $c(4) \leftarrow c(3) \wedge \neg c(4) \wedge c(5)$ | 13 | |
| | | $res(13, 11)$ | $c(4) \leftarrow \neg c(4) \wedge c(5)$ | 14 | 8,11,14 |
| 4 | 110111 \rightarrow 011101 | $R_{c(1)}^{110}$ | $c(1) \leftarrow c(0) \wedge c(1) \wedge \neg c(2)$ | 15 | |
| | | $lg(15, 8)$ | $c(x) \leftarrow c(x-1) \wedge c(x) \wedge \neg c(x+1)$ | 16 | 8,11,14,16 |
| | | $R_{c(2)}^{101}$ | $c(2) \leftarrow c(1) \wedge \neg c(2) \wedge c(3)$ | 17 | |
| | | $lg(17, 14)$ | $c(x) \leftarrow c(x-1) \wedge \neg c(x) \wedge c(x+1)$ | 18 | |
| | | $res(18, 11)$ | $c(x) \leftarrow \neg c(x) \wedge c(x+1)$ | 19 | 8,11,16,19 |
| | | $R_{c(3)}^{011}$ | $c(3) \leftarrow \neg c(2) \wedge c(3) \wedge c(4)$ | 20 | |
| | | $res(20, 19)$ | $c(3) \leftarrow c(3) \wedge c(4)$ | 21 | |
| | | $res(21, 19)$ | $c(3) \leftarrow c(4)$ | 22 | 11,16,19,22 |
| | | $res(8, 22)$ | $c(3) \leftarrow c(3)$ | 23 | 11,16,19,22,23 |

- **Bias II:** The rules are universal for every time step and for any position. This means that the same states of the neighbor cells always implies the same state in the center cell at the next time step.

Combining these two biases, we can adapt **LFIT** to learn dynamics of CAs. Using Bias I, the rule construction process only considers n literals (here $n = 3$) in the neighbors of the cell in the body of a rule. With Bias I, ground resolution is not sufficient to compare non-ground rules with ground rules, for that we need non-ground resolution. We apply anti-instantiation (AI) for getting universal rules with Bias II, whenever a newly added rule R_A^I is not subsumed by any rule in the current program. We can guarantee the soundness of this generalization under Bias II. However, without Bias I, we cannot determine the body literals for construction of each universal rule, so that we must examine the effects from non-neighbor cells too.

TABLE 3.8: LF1T algorithm with Biases I and II on Rule 110 in Torus world

| Step | $I \rightarrow J$ | Operation | Rule | ID | P |
|------|-----------------------------|------------------|--|----|-----------|
| 1 | 000100 \rightarrow 001100 | $R_{c(2)}^{001}$ | $c(2) \leftarrow \neg c(1) \wedge \neg c(2) \wedge c(3)$ | 1 | |
| | | $ai(1)$ | $c(x) \leftarrow \neg c(x-1) \wedge \neg c(x) \wedge c(x+1)$ | 2 | 2 |
| | | $R_{c(3)}^{010}$ | $c(3) \leftarrow \neg c(2) \wedge c(3) \wedge \neg c(4)$ | 3 | |
| | | $ai(3)$ | $c(x) \leftarrow \neg c(x-1) \wedge c(x) \wedge \neg c(x+1)$ | 4 | 2,4 |
| 2 | 001100 \rightarrow 011101 | $R_{c(1)}^{001}$ | $c(1) \leftarrow \neg c(0) \wedge \neg c(1) \wedge c(2)$ | 5 | |
| | | $R_{c(2)}^{011}$ | $c(2) \leftarrow \neg c(1) \wedge c(2) \wedge c(3)$ | 6 | |
| | | $ai(6)$ | $c(x) \leftarrow \neg c(x-1) \wedge c(x) \wedge c(x+1)$ | 7 | |
| | | $res(7, 2)$ | $c(x) \leftarrow \neg c(x-1) \wedge c(x+1)$ | 8 | 8,4 |
| | | $res(7, 4)$ | $c(x) \leftarrow \neg c(x-1) \wedge c(x)$ | 9 | 8,9 |
| | | $R_{c(3)}^{110}$ | $c(3) \leftarrow c(2) \wedge c(3) \wedge \neg c(4)$ | 10 | |
| | | $ai(10)$ | $c(x) \leftarrow \neg c(x-1) \wedge c(x) \wedge \neg c(x+1)$ | 11 | |
| | | $res(11, 9)$ | $c(x) \leftarrow c(x) \wedge \neg c(x+1)$ | 12 | 8,9,12 |
| 3 | 011101 \rightarrow 110111 | $R_{c(1)}^{011}$ | $c(1) \leftarrow \neg c(0) \wedge c(1) \wedge c(2)$ | 13 | |
| | | $R_{c(3)}^{110}$ | $c(3) \leftarrow c(2) \wedge c(3) \wedge \neg c(4)$ | 14 | |
| | | $R_{c(4)}^{101}$ | $c(4) \leftarrow c(3) \wedge \neg c(4) \wedge c(5)$ | 15 | |
| | | $ai(15)$ | $c(x) \leftarrow c(x-1) \wedge \neg c(x) \wedge c(x+1)$ | 16 | |
| | | $res(16, 8)$ | $c(x) \leftarrow \neg c(x) \wedge c(x+1)$ | 17 | 8,9,12,17 |
| 4 | 110111 \rightarrow 011101 | $R_{c(1)}^{110}$ | $c(1) \leftarrow c(0) \wedge c(1) \wedge \neg c(2)$ | 18 | |
| | | $R_{c(2)}^{101}$ | $c(2) \leftarrow c(1) \wedge \neg c(2) \wedge c(3)$ | 19 | |
| | | $R_{c(3)}^{011}$ | $c(3) \leftarrow \neg c(2) \wedge c(3) \wedge c(4)$ | 20 | 8,9,12,17 |

Using Bias I only, **LF1T** in a limited frame of width 4 learns the following rules for Wolfram's Rule 110:

$$\begin{aligned}
c(2) &\leftarrow \neg c(2) \wedge c(3), \\
c(3) &\leftarrow \neg c(2) \wedge c(3), \\
c(3) &\leftarrow c(3) \wedge \neg c(4), \\
c(x) &\leftarrow \neg c(x-1) \wedge c(x+1), \\
c(x) &\leftarrow c(x-1) \wedge c(x) \wedge \neg c(x+1).
\end{aligned} \tag{3.3}$$

Instead, when we use a torus world of length 4 for Wolfram's Rule 110 in **LF1T** with Bias I only, Table 3.7 shows the learning process² and the following NLP is obtained:

$$\begin{aligned}
c(3) &\leftarrow c(3), \\
c(3) &\leftarrow c(4), \\
c(x) &\leftarrow \neg c(x) \wedge c(x+1), \\
c(x) &\leftarrow \neg c(x-1) \wedge c(x+1), \\
c(x) &\leftarrow c(x-1) \wedge c(x) \wedge \neg c(x+1).
\end{aligned} \tag{3.4}$$

²In Tables 3.7 and 3.8, interpretations I and J are represented as configurations, that is, $c(i) \in I$ iff $c(i)$ is true. Operation $lg(R_1, R_2)$ takes the least generalization of R_1 and R_2 with the same pattern, which generalizes the common terms in R_1 and R_2 into variables, and $ai(R)$ takes the anti-instantiation of R .

Both programs (3.3) and (3.4) are quite different from the original rules in Table 3.6. On the other hand, if we use Biases I and II in either a limited frame of width 4 or a torus world of length 4, we get the following NLP (the process is in Table 3.7), which are equivalent to the original transition rule in Table 3.6:

$$\begin{aligned}
 c(x) &\leftarrow c(x) \wedge \neg c(x+1), \\
 c(x) &\leftarrow \neg c(x-1) \wedge c(x), \\
 c(x) &\leftarrow \neg c(x-1) \wedge c(x+1), \\
 c(x) &\leftarrow \neg c(x) \wedge c(x+1).
 \end{aligned} \tag{3.5}$$

In learning an NLP for Rule 110 with Biases I and II, we get interesting generalizations. The NLP obtained from the trace of Rule 110 with **LF1T** becomes more compact in 4 rules, whereas the original transition rule representing the dynamics of this CA in Table 3.6 consists of 5 rules. However, there still exists a redundancy here; we can omit either the second or the third rule from (3.5). But both rules are minimal rules, their body are prime implicant conditions of the head.

3.1.4 Conclusion

We here firstly tackled the induction problem of learning dynamical systems in terms of NLP learning from synchronous state transitions. The proposed algorithm **LF1T** has the following properties:

- Given any state transitions diagram, which is either complete or partial, we can learn an NLP that exactly captures the system dynamics.
- Learning is performed only from positive examples, and produces NLPs that consist only of rules to make literals true.
- Generalization on state transitions rules is done by resolution, in which each rule can only be replaced by a general rule. As a result, an output NLP is always minimal with respect to subsumption among rules.

We have also shown how to incorporate background knowledge and inductive biases, and have applied the framework to learning transition rules of Boolean networks. The results are promising, and implemented programs can be useful for designing the state transitions rules of dynamical systems from a specification of desired or non-desired state transitions diagrams. For instance, a system can be considered to be *robust* if it is tolerant to a perturbation which interferes normal state transitions. Such a transition diagram could be designed as a tree shape, in which its root node corresponds to an attractor, so that any forced state change is eventually recovered

to reach to the attractor [79]. Then we can do reverse engineering to get the corresponding state transitions rules for the Boolean network.

3.2 BDD Algorithms for LF1T

In the previous section we presented the first version of the **LF1T** algorithm. This algorithm has two main weak points. The first one concern its performance: the first implementations were not able to tackle model with more than 15 variables. The second one concern its output: the NLP learned realize the input transitions, but there is no guaranty on the minimality of the rules learned. Lot of improvement could be done regarding the representation of the rules in the implementation. And it is this problem that we tackle first and that we discuss in this section. How to solve the second problem is shown in the next section.

Now we present a new LF1T algorithm based on an efficient data structured inspired from OBDD and Zero-suppressed BDD. A BDD is a canonical representation of a Boolean formula [54, 55]. The novelty of this approach is the integration of LF1T operations into a BDD structure to perform ground resolution. In this approach, one BDD represents a set of rules that have the same head. Figure 3.3 show the evolution of the BDD that represents rules of p in Example 3.2: In this figure, the last schema of step 9 represents a BDD that contains two rules $p \leftarrow p \wedge q$ and $p \leftarrow q \wedge r$ which both have p as their head. The internal nodes of our data structure represent literals, and outgoing edges represent their polarity. In Figure 3.3, the first BDD has one root node which represents the literal p and the edge between its child node q represents the fact that p is positive in the rule $p \leftarrow p \wedge q$.

Like an OBDD [61, 62], our structure respects a total variable ordering: if p, c are two nodes, c is a child of p and l_p, l_c their literals respectively, then $l_p < l_c$. If there is an edge between two nodes p, c that are not neighbors in the ordering, it means that all literals between them are absent from the rules encoded by paths including p and c . Like a ZDD, our BDD structure can have multiple root nodes, but only one leaf; it only represents positive rules. A root node always represents the first literal of one or multiple rules. The leaf node represents the end of all rules; it is the unique child of the last literal of every rule represented by the BDD.

Usual BDD merging operations are not sufficient to perform the generalization operations of LF1T. In LF1T, these operations are equivalent to the use of naïve resolution without P_{old} . In Figure 3.3, the generalization obtained in step 2 can be obtained by usual BDD merging operations: the node r has a positive and negative link to the same node (the leaf) and should be removed according to BDD merging operations. But the generalization obtained by ground resolution on step 9 cannot be obtained by usual BDD merging operations.

To use ground resolution within a BDD structure we need to introduce specific merging operations. These operations have to ensure that the set of rules represented by a BDD is always minimal regarding ground resolution. In Figure 3.3, the last BDD of each learning step respects this notion of minimality.

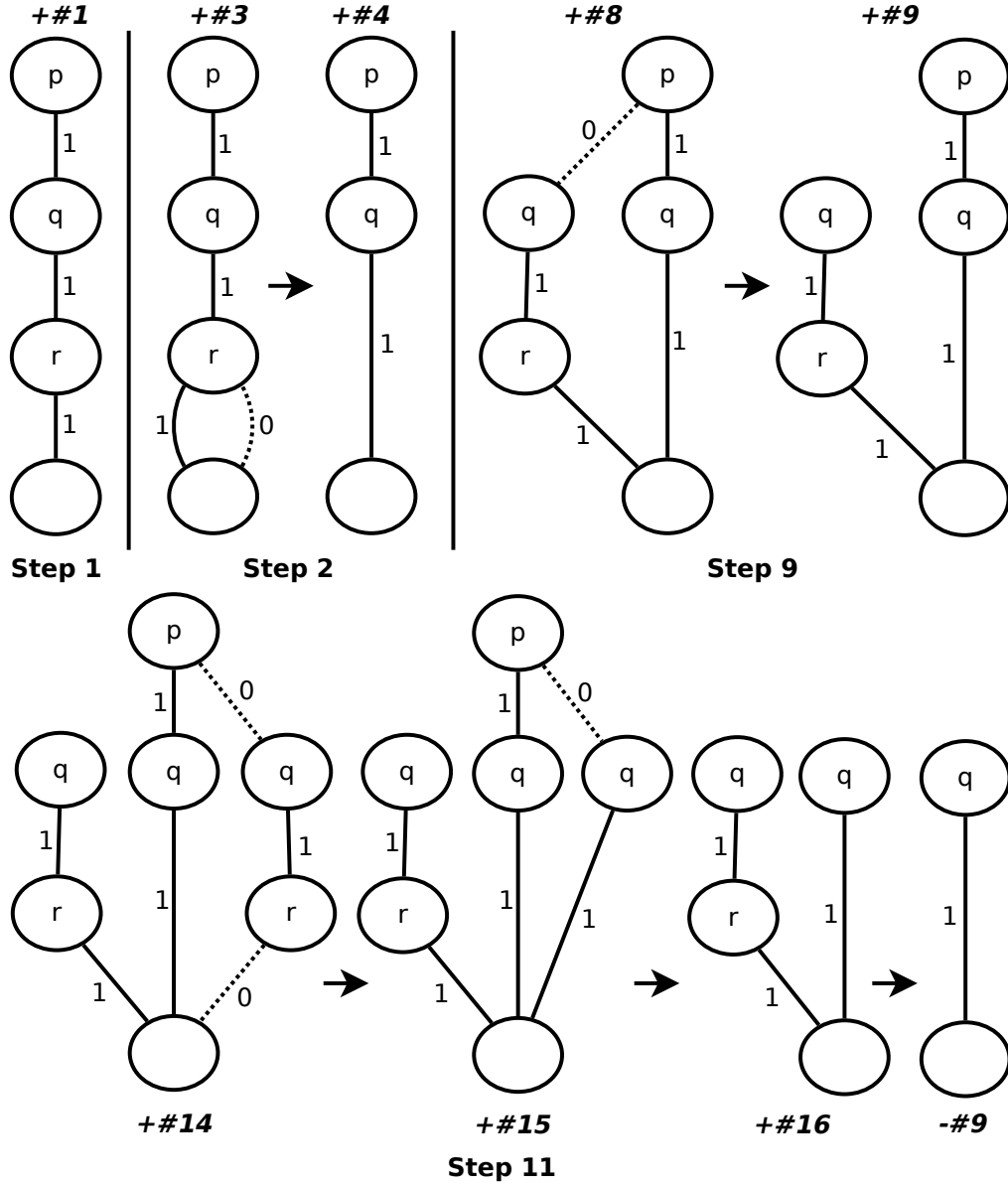


FIGURE 3.3: Evolution of the BDD of p in Example 3.2, edge labelled by 0 represents negation, nodes without parent are roots and the empty node is the leaf. Last schema of each step represents the real state of the BDD; intermediate ones illustrate update operations. Step 1: from (pqr, pq) we learn $p \leftarrow p \wedge q \wedge r$. Step 2: from (pq, p) we learn $p \leftarrow p \wedge q \wedge \neg r$ and by resolution $p \leftarrow p \wedge q$. Step 9: from (qr, pr) we learn $p \leftarrow \neg p \wedge q \wedge r$ and by resolution $p \leftarrow q \wedge r$. Step 11: from (q, pr) we learn $p \leftarrow \neg p \wedge q \wedge \neg r$ which triggers two resolutions and a subsumption to finish with $p \leftarrow q$.

3.2.1 Algorithm

Algorithm 4 describes our adaptation to BDD of the `addRule` operation of LF1T. This algorithm is an application to BDD of the previous version of LF1T based on ground resolution. Whenever a new rule is learned, the corresponding BDD is updated as follows: 1) check if the rule is subsumed, 2) generalize the rule, 3) remove subsumed rules, 4) insert the rule and 5) generalize the BDD. The details of each step is explained as follows.

Algorithm 4 `addRule(R, B)`

```

INPUT: a rule  $R$  and a BDD  $B$ 
 $g$ : a set of rules
// 1) Check if  $R$  is subsumed
for each root node  $r$  of  $B$  do
    if  $r.subsumes(R, 0)$  then return
    end if
end for
// 2) Generalizes  $R$ 
for each root node  $r$  of  $B$  do
    if  $r.generalizes(R, 0)$  then restart the for loop
    end if
end for
// 3) Remove rules subsumed by  $R$ 
 $l :=$  the leaf node of  $B$ 
 $l.clear(R, |R|, true)$ 
// 4) Insert  $R$  into the BDD
 $insert(R, B)$ 
// 5.1) Check generalization by  $R$ 
 $g \leftarrow \emptyset$ 
for each root node  $r$  of  $B$  do
     $r.generalizations(R, 1, g)$ 
end for
// 5.2) Add the generalizations generated by  $R$ 
for each rules  $R_g$  of  $g$  do
     $addRule(R_g)$ 
end for

```

Subsumption (step 1)

To check if a rule is subsumed by a BDD, we have to check whether starting from a root and following the body of the rules allow us to reach the leaf of the BDD. If we reach the leaf then the rule is subsumed. Because we use ground resolution, if a rule is subsumed by the BDD it is useless to search for generalizations of that rule. Checking for such a generalization will only lead to generating a rule that is already in the BDD. Also, it cannot generalize any rules in the BDD: every generalization which can be triggered by this rule has already been found using the rules in the BDD that subsumes it.

Generalization of the new rule (step 2)

To search for generalizations of the rules we use a similar search. However, each time we reach a node representing the current literal l of the rule, we check if the sub-BDDs subsume the

Algorithm 5 $\text{subsumes}(R, n)$ member function of a LFIT-BDD node N

```

1: INPUT: a rule  $R$  and an integer  $n$ 
2: OUTPUT: a Boolean value

3:  $\text{literal}_N$ : literal of the node  $N$ 
4:  $\text{true\_children}$ : list of child nodes linked by a true edge
5:  $\text{false\_children}$ : list of child nodes linked by a false edge
6:  $\text{head}$ : the head literal of  $R$ 
  // 1) Terminal node
7: if  $\text{is\_terminal}()$  AND  $\text{variable} = \text{head}$  then
8:   return true
9: end if
  // 2) End of the rule
10: if  $n > |R|$  then
11:   return false
12: end if
13:  $\text{literal}_R \leftarrow n^{\text{th}}$  literal of  $R$ 
  // 3) LFIT-BDD rules are more generals
14: if  $\text{literal}_R > \text{literal}_N$  then
15:   return  $\text{subsumes}(R, n + 1)$ 
16: end if
17:  $\text{literal}_R \leftarrow n^{\text{th}}$  literal of  $R$ 
  // 4) The rule is more general
18: if  $\text{literal}_R < \text{literal}_N$  then
19:   return false
20: end if
  // 5) Same literal
21: if  $\text{literal}_R$  is positive then
22:    $\text{children} \leftarrow \text{true\_children}$ 
23: else
24:    $\text{children} \leftarrow \text{false\_children}$ 
25: end if
26: for each child node  $c$  of  $\text{children}$  do
27:   if  $c.\text{subsumes}(R, n + 1)$  then
28:     return true
29:   end if
30: end for
31: return false

```

complementary rule on l . If it is the case, we generalize the rule on this literal and restart the check for generalizations with the new rule.

Removal (step 3)

To delete the rules subsumed by the new rule in the BDD, this time we start from the leaf. We follow the parents according to the rule until we check all corresponding parts of the BDD. If we reach the end of the rule, it means that a rule is subsumed. If we do not encounter a node with multiple children, we just have to delete the current node and purge the linked nodes: we recursively delete all parent nodes that have no more children and all children who have no more parents (those poor orphans). Otherwise, we come back to the first node with multiple children we encountered, cut the child edge we followed, and purge the child node in the same way as before.

Algorithm 6 $\text{generalizes}(R, n)$ member function of a LFIT-BDD node N

```

1: INPUT: a rule  $R$  and an integer  $n$ 
2: OUTPUT: a Boolean value
3:  $\text{literal}_N$ : literal of the node  $N$ 
4:  $\text{true\_children}$ : list of child nodes linked by a true edge
5:  $\text{false\_children}$ : list of child nodes linked by a false edge
   // 1) The rule is more general than all rules of the node
6: if  $n > |R|$  then return false
7: end if
   // 2) Terminal node
8: if  $\text{is\_terminal}()$  then return false
9: end if
   // 3) Check generalization on the current node
10:  $\text{literal}_R \leftarrow n^{\text{th}}$  literal of  $R$ 
   // 3.1) The node is more general than the rule
11: while  $\text{literal}_N > \text{literal}_R$  do
12:   if  $\text{subsumes}(R, n)$  then
13:      $R \leftarrow R \setminus \text{literal}_R$  // 3.1.1) The node subsumes the complementary rule
14:     return true
15:   end if
16:    $n \leftarrow n + 1$ 
   // 3.1.2) No more literal to generalize
17:   if  $n > |R|$  then return false
18:   end if
19: end while
   // 3.2) The rule is more general
20: if  $\text{literal}_N < \text{literal}_R$  then return false
21: end if
   // 3.3) The sub-bdd possibly contains the complementary
22:  $\text{same} \leftarrow \text{true\_children}$ 
23:  $\text{opposite} \leftarrow \text{false\_children}$ 
24: if  $\text{literal}_R$  is positive then
25:    $\text{same} \leftarrow \text{false\_children}$ 
26:    $\text{opposite} \leftarrow \text{true\_children}$ 
27: end if
   // 3.3.1) Search for complementary rules
28: for each child node  $c$  of  $\text{opposite}$  do
29:   if  $c.\text{subsumes}(R, n + 1)$  then // Complementary rules is subsumed
30:      $R \leftarrow R \setminus \text{literal}_R$ 
31:     return true
32:   end if
33: end for
   // 4) Search for generalizations on next literal
34: for each child node  $c$  of  $\text{same}$  do
35:   if  $c.\text{generalizes}(R, n + 1)$  then
36:     return true
37:   end if
38: end for
39: return false

```

Insertion (step 4)

All operations we use on our BDDs are based on the manner in which we insert a rule into the structure. First of all, when adding a rule R to a BDD B we assume that R does not subsume and is not subsumed by any rule of B and cannot be generalized by a rule of B using ground

Algorithm 7 $\text{clear}(R, n, \text{can_cut})$ member function of a LFIT-BDD node N

```

1: INPUT:  $R$  a rule,  $n$  an integer and  $\text{can\_cut}$  a Boolean
2: OUTPUT: a Boolean value

3:  $\text{literal}_R$ : the  $n^{\text{th}}$  literal of  $R$ 
4:  $\text{unlink} \leftarrow \text{false}$ 
   // 1) Choice node
5: if  $\#child > 1$  then
6:    $\text{can\_cut} \leftarrow \text{false}$ 
7: end if
   // 2) Check parents
8: for each parent node  $p$  do
9:    $\text{literal}_p \leftarrow$  the literal of  $p$ 
   // 2.1) Parent is more general
10:  if  $\text{literal}_p < \text{literal}_R$  then
11:    if  $n = 1$  AND  $\text{is\_terminal}()$  then
12:      CONTINUE // 2.1.1) Not subsumed
13:    end if
14:    if  $\neg p.\text{clear}(R, n, \text{can\_cut})$  then
15:      CONTINUE
16:    end if
   // 2.1.2) Subsumed
17:    if  $\text{can\_cut}$  then
18:      remove the link with  $p$  and delete  $p$  if it do not has child
19:       $\text{unlink} \leftarrow \text{true}$ 
20:      CONTINUE
21:    end if
22:    return true
23:  end if
   // 2.2) Rule is more general
24:  if  $\text{literal}_p > \text{literal}_R$  then
25:    if  $\neg p.\text{clear}(R, n, \text{can\_cut})$  then
26:      delete  $p$  if it do not has any parent
27:      CONTINUE // 2.2.1) Not subsumed
28:    end if
   // 2.2.2) Subsumed
29:    if  $\text{can\_cut}$  then
30:      remove the link with  $p$  and delete  $p$  if it do not has any child
31:       $\text{unlink} \leftarrow \text{true}$ 
32:      CONTINUE
33:    end if
34:    return true
35:  end if
   // 2.3) Same literal
36:  if  $n > 0$  AND  $\neg p.\text{clear}(R, n - 1, \text{can\_cut})$  then
37:    delete  $p$  if it do not has any parent
38:    CONTINUE
39:  end if
   // 2.3.2) Subsumed
40:  if  $\text{can\_cut}$  then
41:    remove the link with  $p$  and delete  $p$  if it do not has any child
42:     $\text{unlink} \leftarrow \text{true}$ 
43:    CONTINUE
44:  end if
45:  return true
46: end for
47: return false

```

resolution (insured by step 1-3). To add a rule in the BDD we start by searching the common part of the beginning and the end of the body. From the leaf of the BDD, we climb to its parents following the rule from the end. If a parent node has multiple children we do not follow it. Adding a parent to this node will generate more rules than only the one we want to represent. We stop when there is no parent that corresponds to the literal of the rule or when we reach the beginning of the rule. Let us call the last parent reached $last$ and its literal l_{last} ; $last$ will be

connected later to the new nodes created to represent the rule. Then, we search for a root node corresponding to the first literal. If such a root node does not exist, we create a new one, and then we create and link new nodes for all literals $l < l_{last}$ of the rules. Then, $last$ becomes the child of the node most recently created. If a root node corresponds to the first literal of the rule to insert, we follow its children according to the rule body. We stop the descent when no nodes correspond to the rule body, and connect the most recent one we found to $last$. This insertion policy allows us to compile common parts of the rule body to save memory space. It ensures that a node with multiple children have only one parent and cannot have an ancestor with multiple ancestors. In our implementation, this property is exploited to enhance the efficiency of the subsumption and generalization checks of LFIT.

Generalization of BDDs (step 5)

To search the generalizations made by the new rule, we start from the root node. Let l be the current literal we are checking in the rule. When we reach a node whose literal corresponds to l or before it in the ordering, we just have to retrieve all rules subsumed by the rest of the new rules. These rules can all be generalized on the current node. We continue the search for generalizations on the children until we cannot follow the rule anymore. It is necessary to clear the BDD from subsumed rules before this operation in order to avoid a cascade of useless generalizations which lead to the rule we are inserting. In fact, let R_1, R_2 be two rules such that R_1 subsumes R_2 on l . Then R_1 can generalize R_2 on l because R_1 subsumes the complementary of R_2 on l .

Theorem 3.8. *Let n be the size of the Herbrand base $|B|$. Using our dedicated BDD structure the memory complexity as well as the computational complexity of LFIT remain in the same order as the previous algorithm based on ground resolution: , i.e., $O(2^n)$ and $O(4^n)$, respectively.*

Proof. Let n be the size of the Herbrand base $|B|$. This n is also the number of possible heads of rules. Furthermore, n is also the maximum size of a rule, 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 head there are 3^n possible bodies: each literal can either be positive, negative or absent of the body. From these preliminaries we conclude that the size of an NLP $|P|$ learned by LFIT is at most $n \cdot 3^n$. But thanks to ground resolution, $|P|$ cannot exceed $n \cdot 2^n$; in the worst case, P contains only rules of size n where all literals appear and there is only $n \cdot 2^n$ such rules. If P contains a rule with m literals ($m < n$), this rule subsumes 2^{n-m} rules which cannot appear in P . Finally, ground resolution also ensures that P does not contain any pair of complementary rules, so that the complexity is further divided by n ; that is, $|P|$ is bounded by $O(\frac{n \cdot 2^n}{n}) = O(2^n)$.

In our approach, a BDD represents all rules of P that have the same head, so that we have n BDD structures. When $|P| = 2^n$, each BDD represents $2^n/n$ rules of size n and are bound by $O(2^n/n)$, which is the upper bound size of a BDD for any Boolean function [85]. Because BDD

Algorithm 8 $\text{insert}(R, BDD)$

```

1: INPUT: a rule  $R$  and a  $BDD$ 

2: starting: the set of starting nodes of  $BDD$ 
3: literal: first literal of  $R$ 
4: begin, end:  $BDD$  nodes
5:  $n \leftarrow 0$ 
6:  $push \leftarrow false$ 
   // 1) Bottom-up search for common part
7:  $end \leftarrow$  the last ancestor node reached following  $R$  from the corresponding terminal node
   // 2) Fact rule
8: if  $|R| = 0$  then
9:    $starting \leftarrow \{terminalnode\}$ 
10: end if
11:  $begin \leftarrow NULL$ 
   // 2.1) Search common literal within the starting nodes
12: if a node  $r \in starting$  correspond to literal then
13:    $begin \leftarrow r$ 
14: end if
   // 2.2) New starting
15: if  $begin = NULL$  then
16:    $begin \leftarrow$  a new node corresponding to literal
17:    $starting \leftarrow starting \cup \{begin\}$ 
18:    $push \leftarrow true$ 
19: end if
20: current: bdd node pointer
21: make current points on begin
   // 3) Insertion of the rest of the body
22: while  $n \leq |R|$  do
23:    $n \leftarrow n + 1$ 
   // 3.1) Link node reached
24:   if  $n > |R|$  OR the  $n^{th}$  literal of  $R$  is the one of end then
25:     connect current to end according to the polarity of literal
26:     return
27:   end if
28:   literal  $\leftarrow n^{th}$  literal of  $R$ 
   // 3.2) construct new nodes for the rest of the rule
29:   if  $push$  then
30:     create a new node for literal
31:     connect the node to current according to the polarity of literal
32:     make current points on the new node
33:     CONTINUE
34:   end if
   // 3.3) Continue to follow the rule
35:    $next \leftarrow NULL$ 
36:   for each child nodes  $c$  of current according to previous literal polarity do
37:     if  $c$  has only one parent node AND correspond to literal then
38:        $next \leftarrow c$ 
39:       BREAK
40:     end if
41:   end for
   // 3.4) No more common literal
42:   if  $next = NULL$  then
43:      $push = true$ 
44:      $n \leftarrow n - 1$ 
45:     CONTINUE
46:   end if
   // 3.4) // Continue to follow the LF1T-BDD
47:   Make current point on next
48: end while
49: Connect end to begin according to the polarity of literal

```

merges common parts of rules, it is possible that a BDD that represents $2^n/n$ rules needs less than $2^n/n$ memory space. In the previous approach, in the worst case $|P| = 2^n$, whereas in our approach $|P| \leq 2^n$. Our new algorithm still remains in the same order of complexity regarding memory size: $O(2^n)$.

Algorithm 9 $\text{generalizations}(R, n, G)$

```

1: INPUT:  $R$  a rule,  $n$  an integer,  $G$  a list of rules
2: OUTPUT: a Boolean value

3:  $\text{literal}_N$ : node literal
4:  $G', \text{rules}$ : set of rules
   // 1) End of the rule
5: if  $n > |R|$  then return
6: end if
7:  $\text{literal}_R \leftarrow n^{\text{th}}$  literal of  $R$ 
   // 2) Node is more general
8: if  $\text{literal}_N > \text{literal}_R$  then return
9: end if
   // 3) Generalizations are possible on all children
10: if  $\text{literal}_N < \text{literal}_R$  then
11:   for each child node  $c$  do
12:      $\text{rules} \leftarrow$  all rules subsumed by  $R$  in  $c$ 
13:      $G \leftarrow G \cup \{\text{rules}\}$ 
14:   end for
   // 2.2) Retrieve deeper generalizations
15:   for each child node  $c$  do
16:      $G' \leftarrow \emptyset$ 
17:      $c.\text{generalizations}(R, n + 1, G')$ 
18:      $\text{literal} \leftarrow \text{literal}_N$ 
19:     if the link with  $c$  is a negation then
20:        $\text{literal} \leftarrow \neg \text{literal}_N$ 
21:     end if
22:     for each rule  $r$  of  $G'$  do
23:        $G \leftarrow G \cup \{h(r) \leftarrow \text{literal} \wedge \bigwedge_{l \in b(r)} l\}$ 
24:     end for
25:   end for
26:   return
27: end if
   // 3) Same literal
28: for each child node  $c$  do
29:   // 3.1) Search complementary rules
30:   if the link with  $c$  has the same polarity as  $\text{literal}_R$  then
31:      $\text{rules} \leftarrow$  all rules subsumed by  $R$  in  $c$ 
32:      $G \leftarrow G \cup \{\text{rules}\}$ 
33:   else
34:     // 3.2) Check deeper generalizations
35:      $\text{literal} \leftarrow \text{literal}_N$ 
36:     if the link with  $c$  is a negation then
37:        $\text{literal} \leftarrow \neg \text{literal}_N$ 
38:     end if
39:      $G' \leftarrow \emptyset$ 
40:      $c.\text{generalizations}(R, n + 1, G')$ 
41:     for each rule  $r$  of  $G'$  do
42:        $G \leftarrow G \cup \{h(r) \leftarrow \text{literal} \wedge \bigwedge_{l \in b(r)} l\}$ 
43:     end for
44:   end if
45: end for

```

Regarding learning, each operation has its own complexity. Let k be the place of a literal in the variable ordering so that for the starting node literal of a BDD $k = 0$. In our BDD, a node has at most $2 \cdot ((n - k) - 1)$ children: $(n - k) - 1$ positive and negative links to all literals which are superior to k in the ordering. Insertion of a rule is done in polynomial time; in the worst case, we insert a rule where only one literal that differs from the BDD. Because we follow only the first common literals, we have to check at most $2 \cdot ((n - k) - 1)$ links on $n - 1$ nodes, which belongs to $O(n^2)$.

Subsumption as well as generalization checks require exponential time. In the case of subsumption, in the worst case the BDD contains $2^n/n$ rules and the rule is not subsumed by any of them.

That means that we have to check every rule, and each check belongs to $O(n^2)$ so that the whole subsumption operation belongs to $O(n^2 \cdot 2^n/n) = O(2^n)$. To clear the BDD we have to perform the inverse operation. We always have to check the whole BDD, so if the size of the BDD is 2^n then the complexity of the whole clear check also belongs to $O(2^n)$.

To generalize the new rule we have to check if the BDD subsumes one of its complementary rules. Like for subsumption, in the worst case we have to check every rule. A rule can be generalized at most n times; for each generalization we have to check at most n complementary rules, so the complexity of a complete generalization belongs to $O(n^2 \cdot 2^n/n) = O(2^n)$. For the complexity of generalization of BDD rules we consider the inverse problem. In the worst case, every rule of the BDD can be generalized by the new one. Because the new rule does not cover any rules of the BDD, it can generalize each rule of the BDD at most one time. Then, we have at most $2^n/n$ possible direct generalizations on the whole BDD. In the worst case, each of them can be generalized at most $n - 1$ times, and like before, for each generalization we have to check at most n complementary rules. If a rule is generalized n times it means that its body becomes empty, i.e. the rule is a fact, and it will subsume and clear the whole BDD. Then, the complexity of a complete generalization of the BDD belongs to $O(2^n/n \cdot (n - 1) \cdot n) = O(2^n)$.

Each time we learn a rule from a step transition we have to perform these four checks which have a complexity of $O(n^2 + 2^n + 2^n + 2^n) = O(2^n)$. From 2^n state transitions, LF1T can directly infer $n \cdot 2^n$ rules. Learning the dynamics of the entire input implies in the worst case $2^n \cdot 2^n$ operations which belong to $O(4^n)$. Using our dedicated BDD structure the memory complexity as well as the computational complexity of LF1T remains the same order as the previous algorithm based on ground resolution: respectively $O(2^n)$ and $O(4^n)$. \square

3.2.2 Evaluation

In this section, we evaluate our learning methods through experiments. We apply our new LF1T algorithms to learn Boolean networks. Here we run our learning program on the same benchmarks used in [2]. These benchmarks are Boolean networks taken from Dubrova and Teslenko [77], which include those networks for control of flower morphogenesis in *Arabidopsis thaliana*, budding yeast cell cycle regulation, fission yeast cell cycle regulation and mammalian cell cycle regulation. Like in [2], we first construct an NLP $\tau(N)$ from the Boolean function of a Boolean network N where each Boolean function is transformed to a DNF formula. Then, we get all possible 1-step state transitions of N from all $2^{|\mathcal{B}|}$ possible initial states I^0 's by computing all stable models of $\tau(N) \cup I^0$ using the answer set solver `clasp` [86]. Finally, we use this set

| Name | # nodes | # rules | Naïve | Ground | BDD |
|-----------------------------|---------|---------|------------|--------------|--------------|
| <i>Arabidopsis thaliana</i> | 15 | 28 | T.O. | 40.8MB/13.8s | 31.6MB/2.8s |
| Budding yeast | 12 | 54 | 11MB/361s | 4.6MB/0.82s | 3.6MB/0.188s |
| Fission yeast | 10 | 23 | 3.3MB/5.2s | 0.8MB/0.68s | 0.5MB/0.24s |
| Mammalian cell | 10 | 22 | 4.7MB/5.7s | 1MB/0.76s | 0.5MB/0.24s |

TABLE 3.9: Memory use and learning time of **LF1T** for Boolean networks up to 15 nodes with the alphabetical variable ordering

| Name | min/max # rules | Average # rules | time | std deviation rules/time |
|-----------------------------|-----------------|-----------------|-------|--------------------------|
| <i>Arabidopsis thaliana</i> | 29/962 | 227 | 4.31s | 183.03/0.538s |
| Budding yeast | 54/310 | 82 | 0.3s | 41.91/0.019s |
| Fission yeast | 23/45 | 24 | 0.04s | 3.08/0.003s |
| Mammalian cell | 22/22 | 22 | 0.03s | 0/0.007s |

TABLE 3.10: Experimental results of 1000 runs of **LF1T** with random variable orderings

of state transitions to learn an NLP using our LF1T algorithm. Because a run of **LF1T** returns an NLP which can contain redundant rules, the original NLP P_{org} and the output NLP P_{LFIT} can be different, but remain equivalent with respect to state transitions, that is, $T_{P_{org}}$ and $T_{P_{LFIT}}$ are identical functions.

Table 3.9 shows the memory space and time of a single LF1T run in learning a Boolean network for each problem in [77] on a processor Intel Core I7 (3610QM, 2.3GHz) with 4GB of RAM. In the naïve, ground and BDD versions of LF1T the variable ordering is alphabetical. The time limit is set to one hour for each experiment. The gain of memory for the BDD version is up to 50% for the two smaller benchmarks and around 20% for the bigger ones. The main interest of our algorithm is shown by the gain in CPU time. For the *Arabidopsis thaliana* benchmark the input size is quite big: 2^{15} state transitions. Here, naïve version of LF1T reaches the time out (T.O.) of one hour. On this big benchmark, using BDD, we need 80% less CPU time than the previous ground resolution method. These results show that even if the BDD structure does not have a big impact on the whole memory space use, its particular structure allows it to perform LF1T operations faster than in the previous algorithms.

Table 3.10 show more precise experimental results on the BDD version of LF1T. This table shows the minimum, maximum and average number of rules in the output NLP of 1000 runs of LF1T with random variable ordering. The fifth column shows the average learning time and last one is the standard deviation over the number of rules and the one of learning time.

The standard deviation shows that the impact of variable ordering does not affect learning time very much, but it has a significant influence on the rules learned by LF1T. Although those output rules are all minimal with respect to subsumption among them, some are subsumed by original rules. If we consider the original NLP as a kind of optimal NLP in terms of the number of rules, the bigger NLPs learned by our BDD version are local optima where no ground resolutions can be applied among the rules of the NLP. This is because the resolution strategy of LF1T is to

perform resolution only when it produces a generalized rule, so other kinds of resolution are not allowed. For example, from $R_1 = (p \leftarrow p \wedge q)$ and $R_2 = (p \leftarrow \neg q \wedge r)$, $R = (p \leftarrow p \wedge r)$ cannot be obtained in LF1T, since R subsumes neither R_1 nor R_2 . Variable ordering has the same effect on the previous versions of LF1T.

3.2.3 Conclusion

In this section, we proposed a new algorithm for learning from interpretation transitions based on a BDD-like structure. Using this data structure, we can reduce the memory space to represent NLPs learned by LF1T. Analysis of the worst-case computational complexity demonstrated that learning with this method is equivalent to the previous method. However, experimental comparison with previous LF1T algorithms showed that our method outperforms them in practice.

3.3 Learning Prime Implicant Conditions

In this section, our main concern is the minimality of the rules and the NLPs learned by LFIT. Our goal is to learn all minimal conditions that imply a variable to be true in the next state, e.g. all prime implicant conditions. In bioinformatics, for a gene regulatory network, it corresponds to all minimal conditions for a gene to be activated/inhibited. It can be easier and faster to perform model checking on Boolean networks represented by a compact NLP than the set of all state transitions. Knowing the minimal conditions required to perform the desired state transitions, a robot can optimize its actions to achieve its goals with less energy consumption. From a technical point of view, for the sake of memory usage and reasoning time, a small NLP could also be preferred in multi-agent and robotics applications. We use the notion of prime implicant to define minimality of NLP. We consider that the NLP learn by **LFIT** is minimal if the body of each rule constitutes a prime implicant to infer the head.

In [67], prime implicants are define for DNF formula as follows: a clause C , implicant of a formula ϕ , is prime if and only if none of its proper subset $S \subset C$ is an implicant of ϕ . In this work, explanatory induction is considered, while in our approach prime implicants are defined in the LFIT framework. Knowing the Boolean functions, prime implicants could be computed by Tison's consensus method [68] and its variants [69]. The novelty of our approach, is that we compute prime implicants incrementally during the learning of the Boolean function. To the best of our knowledge, there is no previous work that propose a method to compute prime implicant from interpretation of transition. In [67], a method is also proposed to compute prime implicant among DNF formula for explanatory induction is considered, while in our approach prime implicants are defined in the LFIT framework. Our new methods guarantee that the NLPs learned contain only minimal conditions for a variable to be true in the next state.

3.3.1 Formalization

Definition 3.9 (Prime Implicant Condition). Let R be a rule and E a set of state transitions such that R is consistent with E . $b(R)$ is a *prime implicant condition* of $h(R)$ for E if there does not exist another rule R' consistent with E such that R' subsumes R . Let P be a NLP such that $P \cup \{R\} \equiv P$: all models of $P \cup \{R\}$ are models of P and vice versa. $b(R)$ is a *prime implicant condition* of $h(R)$ for P if there does not exist another rule R' such that $P \cup \{R'\} \equiv P$ and R' subsumes R .

Definition 3.10 (Prime Rule). To simplify the rest of this paper, according to Definition 3.9 we will call R a prime rule of E (resp. P) if $b(R)$ is a *prime implicant condition* of $h(R)$ for E (resp. P). For any variable the most general prime rule is the rule with an empty body that states that the variable is always true in the next state.

Example 3.5. Let R_1 , R_2 and R_3 be three rules and E be the set of state transitions of Figure 2.1 as follows: $R_1 = p \leftarrow p \wedge q \wedge r$, $R_2 = p \leftarrow p \wedge q$, $R_3 = p \leftarrow q$. The only rule more general than R_3 is $R' = p$, but R' is not consistent with $(p, \epsilon) \in E$ so that R_3 is a prime rule for E . Since R_3 subsumes both R_1 and R_2 , they are not prime rules of E . Let P be the NLP $\{p \leftarrow p, q \leftarrow p \wedge r, r \leftarrow \neg p\}$, R_3 is a prime rule of P because P realizes E and R_3 is minimal for E .

Definition 3.11 (Prime NLP). Let P be an NLP and E be the state transitions of P , P is a *prime NLP* for E if P realizes E and all rules of P are prime rule for E . We call the set of all prime rules of E the **complete prime NLP** of E .

Example 3.6. Let R_1 , R_2 and R_3 be three rules, E be the set of state transitions of Figure 2.1 and P an NLP as follows: $R_1 = p \leftarrow p \wedge q$, $R_2 = q \leftarrow p \wedge r$, $R_3 = r \leftarrow \neg p$ and $P = \{R_1\} \cup \{R_2\} \cup \{R_3\}$. Since R_1 , R_2 and R_3 are prime rule for E , P the NLP formed of these three rules is a prime NLP of E . There does not exist any other prime rules for E , therefore P is also the complete prime NLP of E .

3.3.2 Learning with full Naïve/ground resolution

The complete prime NLP of a given set of state transitions E can naïvely be obtained by brute force search. Starting from the most general rules that is fact rules, it suffices to generate all maximal specific specialization (Def 2.10) step by step and keep the first ones that are consistent with E . This method implies to check all state transitions for all possible rules that correspond to $O(n \times 3^n \times 2^n) = O(6^n)$ checking operations in the worst case for a Herbrand base of n variables. But it is also possible to do it by extending previous **LF1T** algorithm for the sake of complexity. Here we propose a simple extension of naïve (resp. ground) resolution. In previous algorithms, for each rule learned, only the first least generalization found is kept. Now we consider all possible least generalization and define full naïve (resp. ground) resolution. **LF1T** with full naïve (resp. ground) resolution learn the complete prime NLP that realize the input state transitions.

Definition 3.12 (full naïve resolution and full ground resolution). Let R be a rule and P be a NLP. Let P_R be a set of rule of P such that, for all $R' \in P_R$, $h(R) = h(R')$ and for each R' there exists $l \in b(R)$, $(b(R') \setminus \{\bar{l}\}) = (b(R) \setminus \{l\})$ (resp. $(b(R') \setminus \{\bar{l}\}) \subseteq (b(R) \setminus \{l\})$). The full naïve (resp. ground) resolution of R by P is the set of all possible naïve (resp. ground) resolutions of R with the rules of P : $res_f(R, P) = \{res(R, R') \mid R' \in P_R\}$.

Theorem 3.13 (Completeness and Soundness of full resolution). *Given a set E of pairs of interpretations, **LF1T** with full naïve (resp. ground) resolution is complete and sound for E .*

Proof. According to Theorem 1 (resp. 2) of [2], **LF1T** with naïve (resp. ground) resolution is complete for E . It is trivial that any rules produced by naïve (resp. ground) resolution can be obtained by full naïve (resp. ground) resolution. Then, if P and P' are respectively obtained by naïve (resp. ground) resolution and full naïve (resp. ground) resolution, P' theory-subsumes P . If a program P is complete for E , a program P' that theory-subsumes P is also complete for E . Since P is complete for E by Theorem 1 of [2], P' is complete for E .

All rules that can be produced by naïve (resp. ground) resolution can be obtained by full naïve (resp. ground) resolution. Since all rules produced by naïve (resp. ground) resolution are sound for E (Corollary 1 (resp. 2) of [2]), full naïve (resp. ground) resolution is sound for E . \square

Theorem 3.14 (LF1T with full resolution learn complete prime NLP). *Given a set E of pairs of interpretations, LF1T with full naïve (resp. ground) resolution will learn the complete prime NLP that realize E .*

Proof. Let us assume that **LF1T** with full naïve resolution does not learn a prime NLP of E . If our assumption is correct it implies that there exists R a prime rule for E that cannot be learned by **LF1T** with full naïve resolution. Let \mathcal{B} be the Herbrand base of E .

Case 1: $|b(R)| = |\mathcal{B}|$, R will be directly inferred from a transition $(I, J) \in E$. This is a contradiction with our assumption.

Case 2: $|b(R)| < |\mathcal{B}|$. let l be a literal such that $l \notin b(R)$, according to our assumption, there is a rule R' that is one of the rule $R_1 := h(R) \leftarrow b(R) \cup l$ or $R_2 := h(R) \leftarrow b(R) \cup \bar{l}$ and R' cannot be learned because $res(R_1, R_2) = R$. Recursively, what applies to R applies to R' until we reach a rule R'' such that $|b(R'')| = |\mathcal{B}|$. Our assumption implies that this rule R'' cannot be learned, but R'' will be directly infer from a transition $(I, J) \in E$, this is a contradiction. Since ground resolution can learn all rules learned by naïve resolution, the proof also applies to **LF1T** with full ground resolution. \square

3.3.3 Least Specialization for LF1T

Until now, to construct an NLP, **LF1T** relied on a bottom-up method that generates hypotheses by *generalization* from the most specific clauses or examples until every positive example is covered. Now we propose a new learning method that generate hypotheses by *specialization* from the most general rules until no negative example is covered. Learning by specialization ensures to output the most general valid hypothesis. Specialization is usually considered the dual of generalization in ILP [43, 65, 66]. Where generalization occurs when a hypothesis does not explain a positive example, specialization is used to refine a hypothesis that implies a negative example. The main weak point of the previous **LF1T** algorithms is that the output NLPs depends on variable/transition ordering. Because at each learning step the learned rules by the new **LF1T**

algorithm are minimal according to the state transitions analyzed so far; it implies that our new **LF1T** algorithm can learn more relevant NLPs from partial state transitions than previous ones. Study of the computational complexity of our new method shows that it remains equivalent to the previous version of LF1T. Using examples from the biological literature, we show through experimental results that our specialization method can compete with the previous versions of **LF1T** in practice.

Definition 3.15 (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 . The least specialization $ms(R_1, R_2)$ of R_1 over R_2 is

$$ms(R_1, R_2) = \{h(R_1) \leftarrow b(R_1) \wedge \neg l_i | l_i \in b(R_2) \setminus b(R_1)\}$$

Example 3.7. $R_1 = a \leftarrow a \wedge b \wedge c$ is a least specialization of $R_2 = a \leftarrow a \wedge b$. But R_1 is not a least specialization of $R_3 = a \leftarrow a$ because R_2 is a specialization of R_3 and R_1 is a specialization of R_2 .

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, R be a rule and S be the set of all rules of P that subsume R . The least specialization $ms(P, R)$ of P by R is as follows:

$$ms(P, R) = (P \setminus S) \cup \left(\bigcup_{R_P \in S} ms(R_P, R) \right)$$

Theorem 3.16 (Soundness of least specialization). Let R_1, R_2 be two rules such that R_1 subsumes R_2 . Let S_1 be the set of rules subsumed by R_1 and S_2 be the rules of S_1 that subsume R_2 . The least specialization of R_1 by R_2 only subsumes the set of rules $S_1 \setminus S_2$.

Let P be a NLP and R be a rule such that P subsumes R . Let S_P be the set of rules subsumed by P and S_R be the rules of S_P that subsume R . The least specialization of P by R only subsumes the set of rules $S_P \setminus S_R$.

Proof. :

According to Definition 3.15, the least specialization of R_1 by R_2 is as follows:

$$ms(R_1, R_2) = \{h(R_1) \leftarrow (b(R_1) \wedge \neg b(R_2))\}$$

All rule R of S_2 subsumes R_2 , then according to Definition 4.12 $b(R) \subseteq b(R_2)$. If $ms(R_1, R_2)$ subsumes an R then there exists $R' \in ms(R_1, R_2)$ and $b(R') \subseteq b(R)$. Since $R' \in ms(R_1, R_2)$, there is a $l \in b(R_2)$ such that $\bar{l} \in b(R')$, so that $b(R') \not\subseteq b(R_2)$. Since all $R \in S_2$ subsume R_2 ,

R' cannot subsume any R since R' does not subsume R_2 .

Conclusion 1: the least specialization of R_1 by R_2 cannot subsume any $R \in S_2$.

Let us suppose there is a rule $R' \in S_1$ that does not subsume R_2 and is not subsumed by $ms(R_1, R_2)$. Let l_i be the i^{th} literal of $b(R_2)$, then:

$$ms(R_1, R_2) = \{(h(R_1) \leftarrow (b(R_1) \wedge \bar{l}_i) | l_i \in b(R_2) \setminus b(R_1))\} (1)$$

R' is subsumed by R_1 , so that $R' = h(R_1) \leftarrow b(R_1) \cup S$, with S a set of literal. R' does not subsume R_2 , so that there exists a $l \in b(R_2) \setminus b(R_1)$ such that $\bar{l} \in S$. According to (1), the rule $R'' = h(R_1) \leftarrow b(R_1) \wedge \bar{l}$ is in $ms(R_1, R_2)$. Since R'' subsumes R' and $R'' \in ms(R_1, R_2)$, $ms(R_1, R_2)$ subsumes R' .

Conclusion 2: the least specialization of R_1 by R_2 subsumes all rule of S_1 that does not subsume R_2 .

Final conclusion: the least specialization of R_1 by R_2 only subsumes $S_1 \setminus S_2$.

According to Definition 3.9, the least specialization $ms(P, R)$ of P by R is as follows:

$$ms(P, R) = (P \setminus S_P) \cup \left(\bigcup_{R_P \in S_P} ms(R_P, R) \right)$$

For any rule R_P let S_{R_P} be the set of rules subsumed by R_P and $S_{R_P2} \in S_R$ be the rule of S_{R_P} that subsume R .

The least specialization of R_P by R only subsumes $S_{R_P} \setminus S_{R_P2}$. So that $\bigcup_{R_P \in S_P} ms(R_P, R)$ only subsumes $\left(\bigcup_{R_P \in S_P} S_{R_P} \setminus S_{R_P2} \right) = \left(\bigcup_{R_P \in S_P} S_{R_P} \right) \setminus S_R$. Then $ms(P, R)$ only subsumes the rules subsumed by $(P \setminus S_P) \cup \left(\bigcup_{R_P \in S_P} S_{R_P} \right) \setminus S_R$, that is $S_P \setminus S_R$.

Conclusion: The least specialization of P by R only subsumes $S_P \setminus S_R$. □

3.3.4 Algorithm

Now we present the **LF1T** algorithm based on least specialization. The novelty of this approach is double: first it relies on specialization in place of generalization and most importantly, it guarantees that the output is the complete prime NLP that realize the input transitions, as shown by Theorem 3.18. In this approach the data structures used are the same as the one of the ground version of **LF1T** presented in previous section. Algorithm 10 shows the pseudo-code of **LF1T** with least specialization. Like in previous versions, **LF1T** takes a set of state transitions E as input and outputs an NLP P that realizes E . To guarantee the minimality of the learned NLP, **LF1T** starts with an initial NLP P_0^B that is the most general complete prime NLP of the

Algorithm 10 **LF1T**(E) : Learn the complete prime NLP P of E

```

1: INPUT:  $E \subseteq 2^{\mathcal{B}} \times 2^{\mathcal{B}}$ : (positives) examples/observations
2: OUTPUT: An NLP  $P$  such that  $J = T_P(I)$  holds for any  $(I, J) \in E$ .

3:  $P$  a NLP
4:  $P := \emptyset$ 
   // Initialize  $P$  with the most general rules
5: for each  $A \in \mathcal{B}$  do
6:    $P := P \cup \{A.\}$ 
7: end for
   // Specify  $P$  by interpretation of transitions
8: while  $E \neq \emptyset$  do
9:   Pick  $(I, J) \in E$ ;  $E := E \setminus \{(I, J)\}$ 
10:  for each  $A \in \mathcal{B}$  do
11:    if  $A \notin J$  then
12:       $R_A^I := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$ 
13:       $P := \text{Specialize}(P, R_A^I)$ 
14:    end if
15:  end for
16: end while
17: return  $P$ 

```

Herbrand base \mathcal{B} of E , i.e. the NLP that contains only facts (lines 3-7): $P_0^{\mathcal{B}} = \{p. | p \in \mathcal{B}\}$. Then **LF1T** iteratively analyzes each transition $(I, J) \in E$ (lines 8-16).

For each variable A that **does not appear** in J , **LF1T** infers an **anti-rule** R_A^I (lines 11-12):

$$R_A^I := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$$

Then, **LF1T** uses least specialization to make P consistent with all R_A^I (line 13). Algorithm 21 shows in detail the pseudo code of this operation. **LF1T** first extracts all rules $R_P \in P$ that subsume R_A^I (lines 3-10). It generates the least specialization of each R_P by generating a rule for each literal in R_A^I . Each rule contains all literals of R_P plus the opposite of a literal in R_A^I so that R_A^I is not subsumed by that rule. Then **LF1T** adds in P all the generated rules that are not subsumed by P (line 15-17), so that P becomes consistent with the transition (I, J) and remains a complete prime NLP. When all transitions have been analyzed, **LF1T** outputs P that has become the complete prime NLP of E .

Example 3.8 shows the run of **LF1T** with least specialization on the state transitions of figure 2.1. **LF1T** starts with the most general set of prime rules that can realize E , that is $P = \{p., q., r.\}$. From the transition (pqr, pq) **LF1T** infer the rule $r \leftarrow p \wedge q \wedge r$ that is subsumed by $r. \in P$. **LF1T** then replaces that rule by its least specialization: $ms(r., r \leftarrow p \wedge q \wedge r) = \{r \leftarrow \neg p, r \leftarrow \neg q, r \leftarrow \neg r\}$ Furthermore, P becomes consistent with (pqr, pq) . From (pq, p) **LF1T** infers two rules: $q \leftarrow p \wedge q \wedge \neg r$ and $r \leftarrow p \wedge q \wedge \neg r$, that are respectively subsumed by $q.$ and $r \leftarrow \neg r$. The first rule, $q.$, is replaced by its least specialization: $\{q \leftarrow \neg p, q \leftarrow \neg q, q \leftarrow r\}$. For the second rule, $r.$, its least specialization by $r \leftarrow p \wedge q \wedge \neg r$ generates two rules, $r \leftarrow \neg p \wedge \neg r$

Algorithm 11 $\text{specialize}(P, R)$: specify the NLP P to not subsume the rule R

```

1: INPUT: an NLP  $P$  and a rule  $R$ 
2: OUTPUT: the maximal specific specialization of  $P$  that does not subsumes  $R$ .

3:  $\text{conflicts}$  : a set of rules
4:  $\text{conflicts} := \emptyset$ 
   // Search rules that need to be specialized
5: for each rule  $R_P \in P$  do
6:   if  $R_P$  is conflicting with  $R$  then
7:      $\text{conflicts} := \text{conflicts} \cup R_P$ 
8:      $P := P \setminus R_P$ 
9:   end if
10: end for
   // Revise the rules by least specialization
11: for each rule  $R_c \in \text{conflicts}$  do
12:   for each literal  $l \in b(R)$  do
13:     if  $l \notin b(R_c)$  and  $\bar{l} \notin b(R_c)$  then
14:        $R'_c := (h(R_c) \leftarrow (b(R_c) \cup \bar{l}))$ 
15:       if  $P$  does not subsumes  $R'_c$  then
16:          $P := P \setminus \text{all rules subsumed by } R'_c$ 
17:          $P := P \cup R'_c$ 
18:       end if
19:     end if
20:   end for
21: end for
22: return  $P$ 

```

and $r \leftarrow \neg q \wedge \neg r$. But these rules are respectively subsumed by $r \leftarrow \neg p$ and $r \leftarrow \neg q$ that are already in P . The subsumed rules are not added to P , so that the analysis of (pq, p) results in the specialization of q . and the deletion of $r \leftarrow \neg r$.

Learning continues with similar cases until the last transition (q, pr) where we have a special case. From this transition, **LF1T** infers the rule $q \leftarrow \neg p \wedge q \wedge \neg r$ that is subsumed by P on $R := q \leftarrow \neg p \wedge q \wedge \neg r$. Because $|b(R)| = |\mathcal{B}|$ it cannot be specialized so that P becomes consistent with (q, pr) , **LF1T** just removes R from P .

Example 3.8. Table 4.1 shows the execution of **LF1T** with least specialization on step transitions of figure 2.1 where $pqr \rightarrow pq$ represents the state transitions $(\{p, q, r\}, \{p, q\})$. Introduction of literal by least specialization is represented in bold and rules that are subsumed after specialization are stroked.

Theorem 3.17 (Completeness of **LF1T** with least specialization). *Let $P_0^{\mathcal{B}}$ be the most general complete prime NLP of a given Herbrand base B . Initializing **LF1T** with $P_0^{\mathcal{B}}$, by using least specialization iteratively on a set of state transitions E , **LF1T** learns an NLP that realizes E .*

Proof. :

Let P be an NLP consistent with a set of transitions E' , S_P be the set of rules subsumed by P and a state transitions (I, J) such that $E' \subset E$ and $(I, J) \in E$ but $(I, J) \notin E'$. According to

TABLE 3.11: Execution of **LFIT** with least specialization on step transitions of figure 2.1

| Initialization | $pqr \rightarrow pq$ | $pq \rightarrow p$ | $p \rightarrow \epsilon$ | $\epsilon \rightarrow r$ | $r \rightarrow r$ |
|----------------------|--|---|--|--|---|
| $p.$ $q.$ $r.$ | $p.$ $q.$ $r \leftarrow \neg p.$ $r \leftarrow \neg q.$ $r \leftarrow \neg r.$ | $p.$ $q \leftarrow \neg p.$ $q \leftarrow \neg q.$ $q \leftarrow r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg q.$ $r \leftarrow \neg p \wedge r.$ $r \leftarrow \neg q \wedge r.$ | $p \leftarrow \neg p.$ $p \leftarrow q.$ $p \leftarrow r.$ $q \leftarrow \neg p.$ $q \leftarrow r.$ $q \leftarrow \neg p \wedge \neg q.$ $q \leftarrow \neg q \wedge r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg p \wedge q.$ $r \leftarrow \neg q \wedge r.$ | $p \leftarrow q.$ $p \leftarrow r.$ $p \leftarrow \neg p \wedge q.$ $p \leftarrow \neg p \wedge r.$ $q \leftarrow r.$ $q \leftarrow \neg p \wedge q.$ $q \leftarrow \neg p \wedge r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg q \wedge r.$ | $p \leftarrow q.$ $p \leftarrow p \wedge r.$ $p \leftarrow q \wedge r.$ $q \leftarrow \neg p \wedge q.$ $q \leftarrow p \wedge r.$ $q \leftarrow q \wedge r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg q \wedge r.$ |
| | | | | | |
| | | $qr \rightarrow pr$ | $pr \rightarrow q$ | $q \rightarrow pr$ | |
| | | $p \leftarrow q.$ $p \leftarrow p \wedge r.$ $q \leftarrow p \wedge r.$ $q \leftarrow \neg p \wedge q \wedge \neg r.$ $q \leftarrow \neg p \wedge q \wedge r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg q \wedge r.$ | $p \leftarrow q.$ $p \leftarrow \neg p \wedge q \wedge r.$ $q \leftarrow p \wedge r.$ $q \leftarrow \neg p \wedge q \wedge \neg r.$ $r \leftarrow \neg p.$ $r \leftarrow \neg p \wedge q \wedge r.$ | $p \leftarrow q.$ $q \leftarrow p \wedge r.$ $r \leftarrow \neg p.$ | |
| | | | | $q \leftarrow \neg p \wedge q \wedge \neg r.$ is removed because it cannot be specialized | |

Theorem 3.16, for any rule R_A^I that can be inferred by **LFIT** from (I, J) that is subsumed by P , the least specialization $ms(P, R_A^I)$ of P by R_A^I exactly subsumes the rules subsumed by P except the ones subsumed by R_A^I . Since $|R_A^I|$ is $|\mathcal{B}|$, R_A^I only subsumes itself so that $ms(P, R)$ exactly subsumes $S_P \setminus R_A^I$. Let P' be the NLP obtained by least specialization of P with all R_A^I that can be inferred from (I, J) , then P' is consistent with $E' \cup \{(I, J)\}$.

Conclusion 1: **LFIT** keep the consistency of the NLP learned.

LFIT start with P_0^B as initial NLP. P_0^B is at least consistent with $\emptyset \subseteq E$. According to conclusion 1, initializing **LFIT** with P_0^B and by using least specialization iteratively on the element of E when its needed, **LFIT** learns an NLP that realizes E . \square

Theorem 3.18 (**LFIT** with least specialization output a complete prime NLP). *Let P_0^B be the most general complete prime NLP of a given Herbrand base B . Initializing **LFIT** with P_0^B , by using least specialization iteratively on a set of state transitions E , **LFIT** learns the complete prime NLP of E .*

Proof. :

Let us assume that **LFIT** with least specialization does not learn a prime NLP of E . If our assumption is correct, according to Theorem 3.17, **LFIT** learns a NLP P , that is consistent with E and P is not the complete prime NLP of E . **LFIT** start with P_0^B as initial NLP, P_0^B is the most general complete prime NLP that can cover E .

Consequence 1: LFIT with least specialization can transform a complete prime NLP into an NLP that is not a complete prime NLP.

Let P be the complete prime NLP of a set of state transitions $E' \subset E$ and $(I, J) \notin E'$, such that P is not consistent with (I, J) . Our assumption implies that the least specialization P' of P by the rules inferred from (I, J) is not the complete prime NLP of $E' \cup (I, J)$. According to Definition 3.11, there are two possibilities:

- case 1: $\exists R \in P'$ such that R is not a prime rule of $E' \cup (I, J)$.
- case 2: $\exists R' \notin P'$ such that R' is a prime rule of $E' \cup (I, J)$.

Case 1.1: If $R \in P$, it implies that R is a prime rule of E' and that R is consistent with (I, J) , otherwise R should have been specialized. Because R is not a prime rule of $E' \cup (I, J)$ it implies that there exists a rule R_m consistent with $E' \cup (I, J)$ that is more general than R , i.e. $b(R_m) \subset b(R)$. Then R_m is also consistent with E' , but since R is a prime rule of E' there does not exist any rule consistent with E' that is more general than R . This is a contradiction.

Case 1.2: Now let us suppose that $R \notin P$; then R has been obtained by least specialization of a rule $R_P \in P$ by a rule inferred from (I, J) . It implies that $\exists l \in b(R)$ and $\bar{l} \in I$. If R is not a prime rule of $E' \cup (I, J)$, there exists R_m a prime rule of $E' \cup (I, J)$ and R_m is more general than R . It implies that $l \in R_m$ otherwise R_m is conflicting with (I, J) because it will also subsume R_P that is conflicting with (I, J) . Since R_m is consistent with $E' \cup (I, J)$ it is also consistent with E' . This implies that $\exists R'_m$ a prime rule of E' that subsumes R_m (it can be R_m itself), R'_m also subsumes R . Since P is the complete prime NLP of E' , $R'_m \in P$.

Case 1.2.1: Let suppose that $l \notin b(R'_m)$, since $l \in b(R)$ and R'_m subsumes R then R'_m subsumes R_P because $R = h(R_P) \leftarrow b(R_P) \cup l$. But since R_P is a prime rule of E' it implies that $R'_m = R_P$. In that case it means that R_P subsumes R_m and since $l \in R_m$, $h(R_P) \leftarrow b(R_P) \cup l$ also subsumes R_m . Since $h(R_P) \leftarrow b(R_P) \cup l$ is R , R subsumes R_m and R_m can neither be more general than R nor a prime rule of $E' \cup (I, J)$. This is a contradiction with case 2.

Case 1.2.2: Finally let us suppose that $l \in b(R'_m)$, since R'_m is consistent with E and $l \in I$, R'_m is consistent with $E' \cup (I, J)$. But R'_m subsumes R_m and since R_m is a prime rule of $E' \cup (I, J)$ it implies that $R'_m = R_m$. In that case $R_m \in P$ and because R_m is consistent with (I, J) and R_m subsumes R , LFIT will not add R into P' . This is a contradiction with case 1.

Case 2: Let consider that there exists a $R' \notin P'$ such that R' is a prime rule of $E' \cup (I, J)$. Since $R' \notin P'$, $R' \notin P$ and R' is not a prime rule of E' since P is the complete prime NLP of E' . Then, there exists $R_m \in P$ a prime rule of E' such that R_m subsumes R' and $R_m \notin P'$ since R' is a prime rule of $E' \cup (I, J)$. Then, $b(R') = b(R'_m) \cup S$ with S a non-empty set of literals such that for all $l \in S$, $l \notin b(R_m)$. Since $R_m \notin P'$, there is a rule $R_{h(R_m)}^I$ that can be inferred

from (I, J) and subsumed by R_m . And there is no rule $R'_m \in ms(R_m, R_{h(R_m)}^I)$ that subsumes R' since R' is a prime rule of $E' \cup (I, J)$. Then, for all $l' \in b(R_{h(R_m)}^I)$, $\bar{l}' \notin b(R')$ otherwise there is a R'_m that subsumes R' . Since $|b(R_{h(R_m)}^I)| = \mathcal{B}$, $b(R')$ cannot contain a literal that is not in $b(R_{h(R_m)}^I)$ so that R' subsumes $R_{h(R_m)}^I$. R' cannot be a prime rule of $E' \cup (I, J)$ since R' is not consistent with (I, J) , this is a contradiction.

Conclusion: If P is a complete prime NLP of $E' \subset E$, for any $(I, J) \in E$ **LF1T** with least specialization will learn the complete prime NLP P' of $E' \cup (I, J)$. Since **LF1T** starts with a complete prime NLP that is $P_0^{\mathcal{B}}$, according to Theorem 3.17, **LF1T** will learn a NLP consistent with E , our last statement implies that this NLP is the complete prime NLP of E since **LF1T** cannot specify a complete prime NLP into an NLP that is not a complete prime NLP.

□

Theorem 3.19 (Complexity). *Let n be the size of the Herbrand base $|\mathcal{B}|$. Using least specialization, the memory complexity of **LF1T** remains in the same order as the previous algorithms based on ground resolution, i.e., $O(2^n)$. But the computational complexity of **LF1T** with least specialization is higher than the previous algorithms based on ground resolution, i.e $O(n \cdot 4^n)$ and $O(4^n)$, respectively. Same complexity result for full naïve (resp. ground) resolution.*

Proof. Let n be the size of the Herbrand base $|\mathcal{B}|$ of a set of state transitions E . This n is also the number of possible heads of rules. Furthermore, n is also the maximum size of a rule, 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 head there are 3^n possible bodies: each literal can either be positive, negative or absent from the body. From these preliminaries we conclude that the size of a NLP $|P|$ learned by **LF1T** from E is at most $n \cdot 3^n$. But since a NLP P learned by **LF1T** only contains prime rules of E , $|P|$ cannot exceed $n \cdot 2^n$; in the worst case, P contains only rules of size n where all literals appear and there is only $n \cdot 2^n$ such rules. If P contains a rule with m literals ($m < n$), this rule subsumes 2^{n-m} rules which cannot appear in P . Finally, least specialization also ensures that P does not contain any pair of complementary rules, so that the complexity is further divided by n ; that is, $|P|$ is bounded by $O(\frac{n \cdot 2^n}{n}) = O(2^n)$.

When **LF1T** infers a rule R_A^I from a transition $(I, J) \in E$ where $A \notin J$, it has to compare it with all rules in P to extract conflicting rules. This operation has a complexity of $O(|P|) = O(2^n)$. Since $|b(R_A^I)| = n$, according to Definition 3.9 the least specialization of a rule $R \in P$ can at most generate n different rules. In the worst case all rules of P with $\overline{h(R_A^I)}$ as head subsume R_A^I . There are possibly $2^n/n$ such rules in P , so that **LF1T** generates at most 2^n rules for each R_A^I . For each $(I, J) \in E$, **LF1T** can infer at most n rules R_A^I . In the worst case, **LF1T** can generate $n \cdot 2^n$ rules that are compared with the 2^n rules of P . Thus, construction of an NLP which realizes E implies $n \cdot 2^n \cdot 2^n = n \cdot 4^n$ operations. The same proof applies to **LF1T** naïve (resp. ground) resolution, when **LF1T** infers a rule R_A^I from a transition $(I, J) \in E$ where

$A \in J$. The complexity of learning an NLP from a complete set of state transitions with an Herbrand base of size n is $O(n \cdot 4^n)$. \square

3.3.5 Evaluation

In this section, we evaluate our new learning methods through experiments. We apply our new **LFIT** algorithms to learn Boolean networks. Here we run our learning program on the same benchmarks used in [2] and [53]. These benchmarks are Boolean networks taken from Dubrova and Teslenko [77], which include those networks about control of flower morphogenesis in *Arabidopsis thaliana*, budding yeast cell cycle regulation, fission yeast cell cycle regulation, mammalian cell cycle regulation and T helper cell cycle regulation. Like in previous sections, we first construct an NLP $\tau(N)$ from the Boolean function of a Boolean network N where each Boolean function is transformed into a DNF formula. Then, we get all possible 1-step state transitions of N from all $2^{|\mathcal{B}|}$ possible initial states I^0 's by computing all stable models of $\tau(N) \cup I^0$ using the answer set solver `clasp` [86]. Finally, we use this set of state transitions to learn an NLP using our **LFIT** algorithm. Because a run of **LFIT** returns an NLP which can contain redundant rules, the original NLP P_{org} and the output NLP P_{LFIT} of **LFIT** can be different, but remain equivalent with respect to state transitions, that is, $T_{P_{org}}$ and $T_{P_{LFIT}}$ are identical functions. Regarding the new algorithms, it can also be the case if the original NLP is not a complete prime NLP. For the new versions of **LFIT**, if P_{org} is not a prime complete NLP we will learn a simplification of P_{org} . Table 3.12 shows the memory space and time of a single **LFIT** run in learning a Boolean network for each benchmark on a processor Intel Core I7 (3610QM, 2.3GHz) with 4GB of RAM. It compares memory and run time of the three previous algorithm (naïve, ground and the BDD optimization of the ground version) with their extension to learn complete prime NLP and the new algorithm based on least specialization. For each version of **LFIT** the variable ordering is alphabetical and transition ordering is the one that `clasp` outputs. The time limit is set to two hours for each experiment. Memory is represented in (maximal) number of literal in the NLP learned. Except for **LFIT**-BDD, all implemented algorithms uses the same data structures. That is why even **LFIT** with least specialization cannot compete with the ground-BDD version regarding memory and run time. It is more relevant to compare it to the original implementation of **LFIT** with ground resolution and the new one with full ground resolution.

On Table 3.12 we can observe that, as the number of variable increase, the memory efficiency of least specialization regarding ground version becomes more interesting. Regarding run time, both algorithm have globally equivalent performances. But least specialization ensure that the output is unique in the fact that it is the complete prime NLP of the given input transitions. **LFIT** with full ground resolution also ensure this property, but is much less efficient than least specialization regarding both memory use and run time. On the benchmark, least specialization is

| Algorithm | Mammalian (10) | Fission (10) | Budding (12) | Arabidopsis (16) | T helper (23) |
|----------------------|--------------------|--------------------|------------------|-------------------|-------------------|
| Naïve | 142 118/4.62s | 126 237/3.65s | 1 147 124/523s | T.O. | T.O. |
| Ground | 1036/ 0.04s | 1218/ 0.05s | 21 470/0.26s | 271 288/4.25s | T.O. |
| Ground-BDD | 180/0.24s | 147/0.24s | 541/0.19s | 779/2.8s | 611/3360s |
| Full Naïve | 377 539/29.25s | 345587/24.03s | T.O. | T.O. | T.O. |
| Full Ground | 1066/0.24s | 1178/0.23s | 23 738/4.04s | 399 469/111s | T.O. |
| Least Specialization | 375/0.06s | 377/0.08s | 641/0.35s | 2270/5.28s | 3134/5263s |

TABLE 3.12: Memory use and learning time of **LF1T** for Boolean networks benchmarks up to 23 nodes in the same condition as in [2]

| Benchmark | Nodes | min/max # rules | min/max time |
|-----------|-------|-----------------|--------------|
| Budding | 12 | 54/54 | 1.22/3190 |
| Fission | 10 | 24/24 | 0.20/0.69 |
| Mammalian | 10 | 22/22 | 0.17/0.61 |

TABLE 3.13: Results of 1000 runs of **LF1T with least specialization** for Boolean networks benchmarks: random transition orderings

| Algorithm | Scalability | Run Time | Memory | Output | Variable Order | Transition Order |
|----------------------|-------------|----------|--------|--------|----------------|------------------|
| Naïve | 12 | -- | -- | -- | -- | -- |
| Ground | 16 | ++ | ++ | -- | -- | -- |
| Ground-BDD | 23 | +++ | +++ | -- | -- | -- |
| Full Naïve | 10 | -- | -- | +++ | +++ | +++ |
| Full Ground | 16 | -- | -- | +++ | +++ | -- |
| Least Specialization | 23 | ++ | ++ | +++ | +++ | -- |

TABLE 3.14: Properties of the different **LF1T** algorithms

respectively 75%, 65%, 91% and 95% faster. Least specialization version also succeed to learn the t-helper benchmark (23 variables) in 1 hour and 21 minutes. The main interest of using least specialization is that it guarantees to obtain a unique NLP that contains all minimal conditions to make a variable true. Previous versions of **LF1T** do not have this property and experimental results showed that their output is sensitive to variable ordering and especially transition ordering. For a given set of state transitions E , the output of **LF1T** with least specialization is always the same whatever the variable ordering or transition ordering. It is easy to see that variable ordering has no impact on both learning time and memory use of the new versions of **LF1T** since they consider all generalizations/specializations. But transition ordering has a bigger impact on the learning time of the new version of **LF1T** compared to others as shown by the results of Table 3.13.

Table 3.14 show the comparison of the different versions of **LF1T** regarding scalability, run time and memory use. It also highlight the relative degree of sensitivity of each algorithm regarding variable and transition ordering.

Regarding run time and memory usage, the Ground-BDD version is the best non-minimal algorithm of the framework we propose. But variable ordering have an impact on the run time of this algorithm, where it is not the case for the algorithm with minimal guaranties. The Least Specialization approach is the most efficient algorithm we have that guaranty minimal rules in

output. But transition ordering impacts both run time and memory, it is very significant on the three algorithm that guaranty minimal rules.

3.4 Conclusion and Future Work

We proposed several algorithms for learning Boolean synchronous deterministic system from interpretation transitions. Given any state transition diagram we can now learn an NLP that exactly captures the system dynamics. Learning is performed only from positive examples, and produces NLPs that consist only of rules to make literals true. Consistency of state transition rules is achieved and minimality of rules is guaranteed. As a result, given any state transition diagram E , **LF1T** always learns a unique NLP that contains all prime rules that realize E . It implies that the output of **LF1T** is not sensitive to variable ordering or transition ordering. But, experimental results showed that the algorithm is sensitive to input transitions ordering regarding run time.

Chapter 4

Framework Extensions

In the previous chapter we introduced the basis of our learning framework. Until now our modeling and algorithms require the variable of the system to be Boolean and capture only synchronous deterministic semantics. In this chapter we extend our methods to address more complex systems structure and dynamics. First, we propose a modelization of system with delayed influences as logic program and algorithms to learn those semantics in section 4.1. Later, in section 4.2, we get rid of the Boolean limitation of the variable by providing a modelization of multi-valued systems. With this modelisation we extend our algorithm dedicated to delayed system in section 4.3. Then, we provide the simple extension to asynchronous semantics in section 4.4. Finally, in section 4.5 we propose a modelisation of non-deterministic and probabilistic system, as well as algorithm to capture their dynamics from uncertain state transitions.

The algorithm that learns delayed influences presented in section 4.1 has been published in the journal *Frontiers in Bioengineering and Biotechnology* [73]. The formalization of multi-valued systems of section 4.2 and the algorithm that learns probabilistic system in section 4.5 have been accepted as a technical communication paper in the *31st International Conference on Logic Programming (ICLP 2015)*.

4.1 Delayed Systems

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 [70] or the DNA damage repair [71]. Social interactions too may depend on the behaviors history of the agents at stake [87]. 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 [72] 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. [88]: 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 $\{t, t - 1, \dots, t - (k - 1)\}$. In this chapter, we will consider Markov(k) model and discuss new learning algorithms.

We now extend our framework by designing an algorithm that takes multiple sequences of state transitions as input and builds a normal logic program that captures the delayed dynamics of a Markov(k) system. While the previous algorithm dealt only with 1-step transitions (i.e., we assume the state of the system at time t depends only of its state at time $t - 1$), we propose here an approach that is able to consider k -step transitions (sequence of at most k state transitions).

4.1.1 Formalization

A Markov(1) system can be represented by a deterministic state transitions diagram, like the one in Figure 2.1. 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 Boolean network 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 Boolean network may then depend on its (at most) k previous states.

Definition 4.1 (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:

$$\mathcal{B}_k = \bigcup_{i=1}^k \{v_{t-i} \mid v \in \mathcal{B}\}$$

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

According to Definition 4.16, given a propositional atom v , v_j 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 4.2 (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 head, 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 a Boolean value modeling the fact that it is lower or greater than a threshold.

Example 4.1. Let R_1 and R_2 be two rules, $R_1 = a \leftarrow b_{t-1} \wedge b_{t-2}$, $R_2 = b \leftarrow a_{t-2} \wedge \neg b_{t-2}$. The logic program $S = \{R_1, R_2\}$ is a Markov(2) system, i.e., the state of the system depends on the two previous states. The value of a is true at time step t only if b was true at $t - 1$ and $t - 2$. The value of b is true at time step t only if a was true at $t - 2$ and b was false at $t - 2$. The atoms that appear in the head of the rules of S is $\{a, b\}$. \mathcal{B}_1 represents these atoms from time step $t - 1$: $\mathcal{B}_1 = \{a_{t-1}, b_{t-1}\}$ and \mathcal{B}_2 represents these atoms from time step $t - 2$: $\mathcal{B}_2 = \{a_{t-1}, b_{t-1}, a_{t-2}, b_{t-2}\}$.

In the following definitions, we refer to \mathbb{N} as the set of all natural numbers.

Definition 4.3 (Trace of execution). Let \mathcal{B} be the atoms that appear in the head of the rules of a Markov(k) system S . A trace of execution T is a finite sequence of states of S : $T = (S_0, \dots, S_n)$, $n \geq 1, \forall i \in \mathbb{N}, i \leq n, S_i \in 2^{\mathcal{B}}$. For all $j \in \mathbb{N}$, we define:

$$prev(i, j, T) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0, \\ (S_{i-j-1}, \dots, S_{i-1}) & \text{if } j + 1 \leq i \\ (S_0, \dots, S_{i-1}) & \text{otherwise.} \end{cases}$$

We also define $prev(i, T) = prev(i, n, T)$ and $next(i', T) = S_{i'+1}$, $i' \in \mathbb{N}, i' < n$.

We denote by $|T|$ the size of the trace, that is the number of elements of the sequence. A sub-trace of size m of a trace of execution T is a sub-sequence of consecutive states of T of size

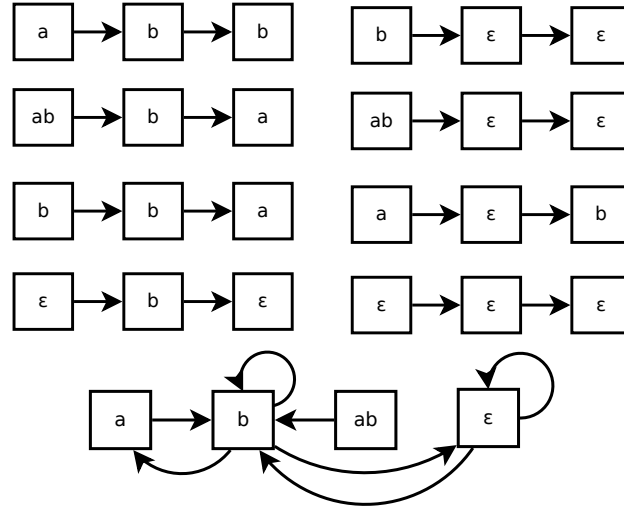


FIGURE 4.1: Eight traces of executions of the system of Example 4.1 (left) and the corresponding state transitions diagram (right)

m , where $m \in \mathbb{N}, 1 < m \leq |T|$. In the following, we will also denote $T = (S_0, \dots, S_n)$ as $T = S_0 \rightarrow \dots \rightarrow S_n$.

Example 4.2. Let $T_1 = a \rightarrow b \rightarrow a$ be a trace of execution. T_1 is a trace of size 2 and $a \rightarrow b$ and $b \rightarrow a$ are sub-traces of size 1 of T_1 .

Definition 4.4 (Consistent traces). Let $T = (S_0, \dots, S_n)$ and $T' = (S'_0, \dots, S'_m)$ be two traces of execution. T and T' are k -consistent, with $k \in \mathbb{N}$, iff $\forall i, j \in \mathbb{N}, i < n, j < m, S_i = S_j$ and $next(i, T) \neq next(j, T') \implies prev(i, k, T) \neq prev(j, k, T')$. T and T' are said consistent iff they are $\max(n, m)$ -consistent.

As shown in Figure 4.4, a Markov(k) system may seem non-deterministic when it is represented by a state transitions diagram (right part of the figure). That is because such state transitions diagram only represents 1-step transitions. In this example, the transition from the state b is not Markov(1): the next state can be either a, b or ϵ . But it can be Markov(2), because all traces of size 2 of Figure 4.4 are consistent.

Definition 4.5 (k -step interpretation transitions). 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 w.r.t $P, k' \geq k$. A k -step interpretation transition is a pair of interpretations (I, J) where $I \subseteq \mathcal{B}_k$ and $J \subseteq \mathcal{B}$.

Example 4.3. The trace $ab \rightarrow b \rightarrow a$ can be interpreted in the three following ways:

- $(a_{t-2}b_{t-2}b_{t-1}, a)$: the 2-step interpretation transition that corresponds to the full trace $ab \rightarrow b \rightarrow a$.
- $(a_{t-1}b_{t-1}, b)$: the 1-step interpretation transition corresponding to the sub-trace $ab \rightarrow b$.
- (b_{t-1}, a) : the 1-step interpretation transition that corresponds to the sub-trace $b \rightarrow a$.

Definition 4.6 (Extended Consistency). Let R be a rule and (I, J) be a k -step interpretation transition. R is *consistent* with (I, J) iff $b^+(R) \subseteq I$ and $b^-(R) \cap I = \emptyset$ imply $h(R) \in J$. Let T be a sequence of state transitions, R is consistent with T if it is consistent with every k -step interpretation transitions of T . Let O be a set of sequences of state transitions, R is consistent with O if R is consistent with all $T' \in O$.

4.1.2 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 . The learned influences can be at most k -step relations, where k is the size of the longest trace of O . The main idea is to extract n -step interpretation transitions, $1 \leq n \leq k$, from the traces of executions of the system. Transforming the traces into pairs of interpretations allows us to use minimal specialization [64] to iteratively learn the dynamics of the system.

LFkT:

- **Input:** A set of traces of execution E of a Markov(k) system S .
- Step 1: Initialize k logic programs with facts rules.
- Step 2: Convert the input traces of executions into interpretation transitions.
- Step 3: Revise iteratively the logic programs by all interpretation transitions using minimal specialization.
- Step 4: Merge all logic programs into one.
- **Output:** The rules of S which generated E .

The idea of the algorithm is to start with the most general rules (Algorithm 17 1.6-10) and use specialization to make them consistent with the input observations (algo.2). The algorithm analyzes each interpretation transition one by one and revises the learned rules when they are not consistent (Algorithm 17 1.13-23). After analyzing all interpretation transitions, the programs that have been learned are merged into a unique logic program (Algorithm 17 1.24-29). This operation ensures that the rules outputted are consistent with all observations. Finally, **LFkT** outputs a logic program that realizes all consistent traces of execution of O . We now provide the detailed explanation of each step of the algorithm.

Step 1: The algorithm starts with a vector of size k where each element is the logic programs $\{p \mid p \in \mathcal{B}\}$. In the following, we will call a n -step rule a rule R such that $\forall v_{t-i} \in b(R), 1 \leq$

Algorithm 12 LFKT(O) : Learn the most general rules that explain E

```

1: INPUT:  $O$  a set of sequences of state transitions (traces of executions)
2: OUTPUT: The logic program that realizes the transitions of  $O$ .

3:  $P'$  a vector of set of rules
4:  $E$  a vector of set of pairs of interpretations  $(I, J)$ 
5:  $max_k :=$  the size of the longest trace of  $O$ 
   // 1) Initialize  $P'$  with  $max_k$ , the logic program that contains the most general rules
6: for each  $k$  from 1 to  $max_k$  do
7:    $P'_k := \emptyset$ 
8:   for each  $A \in \mathcal{B}$  do
9:      $P'_k := P'_k \cup \{A.\}$ 
10:  end for
11: end for
12: // 2) Extract interpretation from trace of executions
13:  $E := \text{interpret}(O)$ 

14: // 3) Specify  $P'$  by the interpretation of the trace of executions
15: for each  $k$  from 1 to  $max_k$  do
16:    $E_k :=$  the  $k^{th}$  set of interpretation of  $E$ 
17:   while  $E_k \neq \emptyset$  do
18:     Pick  $(I, J) \in E_k$ ;  $E_k := E_k \setminus \{(I, J)\}$ 
19:     for each  $A \in \mathcal{B}$  do
20:       if  $A \notin J$  then
21:          $R_A^I := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$ 
22:          $P'_k :=$  the  $k^{th}$  set of rule of  $P'$ 
23:          $P'_k := \text{Specialize}(P'_k, R_A^I)$ 
24:       end if
25:     end for
26:   end while
27: end for

28: // 4) Merge the programs into a unique logic program
29:  $P := \emptyset$ 
30: for each  $k$  from 1 to  $max_k$  do
31:    $P'_k :=$  the  $k^{th}$  set of rule of  $P'$ 
32:   Remove from  $P'_k$  all rules that do not contain any literal of the form  $v_{t-k}$ 
33:    $P := P \cup P'_k$ 
34: end for
35: return  $P$ 

```

$i \leq n$. The idea is to learn rules independently for each possible k -step relation: 1-step rules, 2-step rules, \dots , k -step rules. The rules learned from 1-step interpretations will go into the 1-step program, the rules learned from 2-step interpretations will go into the 2-step program and so on. These different programs are merged at the end to constitute a logic program that realizes all consistent traces of O .

Step 2: In order to use minimal specialization, we need to convert the input traces of execution into interpretation transitions. This conversion is done by the function **interpret**, whose pseudo code is given in Algorithm 20. It extracts all k -step interpretations from each trace $T \in O$. It can be done by extracting and converting all sub-traces of T into corresponding interpretations.

This way it produce one $|T|$ -step interpretation, one $|T| - 1$ interpretation, \dots , one 1-step interpretation. The function outputs them as a vector of set of interpretation transitions E , where each set E_i corresponds to interpretation of sub-traces of size i .

Algorithm 13 `interpret(O)` : Extract interpretations transition from traces

```

1: INPUT:  $O$  a set of sequences of state transitions (traces of executions)
2: OUTPUT:  $E$  a vector of set of pairs of interpretations  $(I, J)$ 

3:  $E := \emptyset$ 
   // Extract interpretations
4: for each sequence  $T \in O$  do
5:   for each  $k$  from  $|T|$  to 1 do
6:     for each sub-trace  $T'$  of size  $k$  in  $T$  do
7:        $s_k :=$  the  $k^{th}$  state of  $T'$ 
8:        $I := \emptyset$ 
9:       for each state  $s_{k'}$  before  $s_k$  in  $T'$  do
10:         $i := k - k'$ 
11:        for each atom  $a \in s_{k'}$  do
12:           $I := I \cup \{a_{t-i}\}$ 
13:        end for
14:         $E_k :=$  the  $k^{th}$  set of interpretation of  $E$ 
15:         $E_k := E_k \cup \{(I, s_k)\}$ 
16:      end for
17:    end for
18:  end for
19: end for
20: return  $E$ 
```

Step 3: The algorithm iteratively learns from each set of pairs of interpretations $E_i \in E$. Now it only needs to apply the *LFIT* method of [64] on each set E_i by analyzing each pair of interpretations $(I, J) \in E_i$. For each variable A that **does not appear** in J , it infers an **anti-rule** $R_A^I := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (B_i \setminus I)} \neg C_j$, where B_i is the i -step atoms of E_i , i.e. all atoms that can appear in a rule of E_i . Then, minimal specialization is used to make the corresponding logic program P'_i consistent with R_A^I . Algorithm 27 shows the pseudo code of this operation. In the function **specialize**, it first extracts all rules $R_P \in P$ that subsumes R_A^I . It generates the minimal specialization of each R_P by generating a rule for each literal in R_A^I . Each rule contains all literals of R_P plus the opposite of a literal in R_A^I so that R_A^I is not subsumed 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) . When all transitions have been analyzed, *LFIT* outputs P that has become the logic program that realizes E .

Step 4: After analyzing all interpretation transitions, the programs that have been learned are merged into a unique logic program. This operation guarantees that the rule outputted are consistent with all input traces of executions. If a rule is not consistent with a trace of execution, it has to be deleted. It can be checked by comparing each rule with other logic programs. If a n -step rule R is more general than a n' -step rules R' , $n' < n$, then R is not consistent with the observations from which R' has been learned. To avoid this case, we just need to remove

Algorithm 14 $\text{specialize}(P, R)$: specialize the logic program P to not subsume the rule R

```

1: INPUT: a logic program  $P$  and a rule  $R$ 
2: OUTPUT: the minimal specific specialization of the rule of  $P$  by  $R$ .

3:  $\text{conflicts}$  : a set of rules
4:  $\text{conflicts} := \emptyset$ 
   // Search rules that need to be specialized
5: for each rule  $R_P \in P$  do
6:   if  $R_P$  subsumes  $R$  then
7:      $\text{conflicts} := \text{conflicts} \cup R_P$ 
8:      $P := P \setminus R_P$ 
9:   end if
10: end for
   // Revise the rules by minimal specialization
11: for each rule  $R_c \in \text{conflicts}$  do
12:   for each literal  $l \in b(R)$  do
13:     if  $l \notin b(R_c)$  and  $\bar{l} \notin b(R_c)$  then
14:        $R'_c := (h(R_c) \leftarrow (b(R_c) \cup \bar{l}))$ 
15:       if  $P$  does not subsume  $R'_c$  then
16:          $P := P \setminus \{R \mid R \text{ is subsumed by } R'_c\}$ 
17:          $P := P \cup R'_c$ 
18:       end if
19:     end if
20:   end for
21: end for
22: return  $P$ 

```

n -step rules that have no v_n variable. Finally, **LFkT** outputs a logic program that realizes all consistent traces of executions of O . If O is a set of traces of execution of a Boolean network, the logic program outputted by **LFkT** represents the Boolean functions of each variables. For each variable, it corresponds to the conditions over the k previous step to make it active at $t + 1$.

Theorem 4.7 (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. Let V be the vector of interpretation transition extracted from O by **LFkT** (Algorithm 20). According to Theorem 4 of [64], initializing *LFIT* with $\{p \mid p \in \mathcal{B}\}$, by using minimal specialization iteratively on a set of interpretation transitions E , we obtain a logic program P that realizes E . Since **LFkT** use the same method as *LFIT* 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 $\{p \mid p \in \mathcal{B}\}$ with all anti-rule of v_n (non consistent rule). According to Theorem 3 of [64], 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 contains 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 learn from $v_1 \in V$, and let $P = p'_1$. Let R' be all rule 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 . So that, using O as input, **LFkT** outputs a logic program that realizes all consistent traces of O . \square

Theorem 4.8 (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 Markov(k) system with respect to P . Let O be a set of trace of execution of S . The complexity of learning S from O with **LFkT** is respectively: $O(2^{nk})$ for memory and $O(\sum_{T \in O} |T| \cdot n2^{nk})$ for runtime.*

Proof. $n = |\mathcal{B}|$ is the number of possible heads of rules of S . $nk = |\mathcal{B}_k|$ 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 3^{nk} possible bodies: each literal can either be positive, negative 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 $|S| = n \cdot 3^{nk}$. But thanks to minimal specialization, $|S|$ cannot exceed $n \cdot 2^{nk}$; in the worst case, S contains only rules of size nk where all literals appear and there is only $n \cdot 2^{nk}$ such rules. If S contains a rule with m literals ($m < nk$), this rule subsumes 2^{nk-m} rules which cannot appear in S . Finally, minimal specialization also ensures that S does not contain any pair of complementary rules, so that the complexity is further divided by nk ; that is, $|S|$ is bounded by $O(\frac{n \cdot 2^{nk}}{nk}) = O(\frac{2^{nk}}{k})$. To learn S , **LFkT** needs to store k programs P_i that are Markov(i) system with respect to P , $1 \leq i \leq k$.

Conclusion 1: the memory use of **LFkT** is $O(\sum_{i=1}^k |P_i|) = O(k \cdot \frac{2^{nk}}{k}) = O(2^{nk})$.

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_A^I for each $A \in \mathcal{B}$, $A \notin J$. **LFkT** compares each R_A^I with all rules of each programs P_i . There is atmost $|\mathcal{B}|$ anti-rules that can be inferred from (I, J) by

LFkT and the size of each program P_i is bound by $O(\frac{2^{nk}}{k})$. Then, the complexity of learning one trace of execution $T \in O$ with **LFkT** is $O(|T| \cdot |\mathcal{B}| \cdot k|P_i|) = O(|T| \cdot n \cdot k \frac{2^{nk}}{k}) = O(|T| \cdot n 2^{nk})$.

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

4.1.3 Running example

Table 4.1 shows the execution of **LFkT** on traces of figure 4.4 where $(a_2 b_2 b_1, a)$ represents the interpretation of the trace $ab \rightarrow b \rightarrow a$. Introduction of literal by minimal specialization is represented in bold and rules that are subsumed after specialization are stroked. For the sake of readability, here a_1 and a_2 respectively correspond to a_{t-1} and a_{t-2} .

| Initialization | | $a \rightarrow b \rightarrow b$ | | $ab \rightarrow b \rightarrow a$ | |
|---|--|---|--|---|---|
| 2-step NLP | 1-step NLP | $(a_2 b_1, b)$ | (a_1, b) | $(a_2 b_2 b_1, a)$ | $(a_1 b_1, b)$ |
| $a.$ $b.$ | $a.$ $b.$ | $a \leftarrow \neg a_2.$ $a \leftarrow b_2.$ $a \leftarrow a_1.$ $a \leftarrow \neg b_1.$ $b.$ | $a \leftarrow \neg a_1.$ $a \leftarrow b_1.$ $b.$ | $a \leftarrow \neg a_2.$ $a \leftarrow b_2.$ $a \leftarrow a_1.$ $b \leftarrow \neg a_2.$ $b \leftarrow \neg b_2.$ $b \leftarrow a_1.$ $b \leftarrow \neg b_1.$ | $a \leftarrow \neg a_1, \neg b_1.$ $a \leftarrow a_1, b_1.$ $b.$ (b_1, a) $a \leftarrow \neg a_1, \neg b_1.$ $b \leftarrow a_1.$ $b \leftarrow \neg b_1.$ |
| $b \rightarrow b \rightarrow a$ | | $\epsilon \rightarrow \epsilon \rightarrow \epsilon$ | | $b \rightarrow \epsilon \rightarrow \epsilon$ | |
| $(b_2 b_1, a)$ | (b_1, b) | (ϵ, ϵ) | (ϵ, ϵ) | (b_2, ϵ) | (b_1, ϵ) |
| $a \leftarrow \neg a_2.$ $a \leftarrow b_2.$ $a \leftarrow \neg a_1.$ $a \leftarrow \neg b_1.$ $b \leftarrow \neg a_2, \neg b_2.$ $b \leftarrow \neg a_2, a_1.$ $b \leftarrow \neg a_2, \neg b_1.$ $b \leftarrow \neg b_2.$ $b \leftarrow a_1.$ $b \leftarrow \neg b_1.$ | $a \leftarrow \neg a_1, \neg b_1.$ $b \leftarrow a_1.$ $b \leftarrow \neg b_1.$ (b_1, a) $a \leftarrow \neg a_1, \neg b_1.$ $b \leftarrow a_1.$ $b \leftarrow \neg b_1.$ | $a \leftarrow \neg a_2, \neg b_2.$ $a \leftarrow \neg a_2, a_1.$ $a \leftarrow \neg a_2, b_1.$ $a \leftarrow b_2.$ $a \leftarrow a_1.$ $a \leftarrow a_2, \neg b_1.$ $a \leftarrow \neg b_2, \neg b_1.$ $a \leftarrow a_1, \neg b_1.$ $b \leftarrow a_2, \neg b_2.$ $b \leftarrow \neg b_2, a_1.$ $b \leftarrow \neg b_2, b_1.$ $b \leftarrow a_1.$ $b \leftarrow a_2, \neg b_1.$ $b \leftarrow b_2, \neg b_1.$ $b \leftarrow a_1, \neg b_1.$ | $a \leftarrow \neg a_1, \neg b_1.$ $b \leftarrow a_1.$ $b \leftarrow \neg a_1, \neg b_1.$ (ϵ, ϵ) $b \leftarrow a_1.$ | $a \leftarrow \neg a_2, b_1.$ $a \leftarrow a_2, b_2.$ $a \leftarrow \neg b_2, a_1.$ $a \leftarrow b_2, b_1.$ $a \leftarrow a_1.$ $a \leftarrow a_2, \neg b_1.$ $b \leftarrow a_2, \neg b_2.$ $b \leftarrow \neg b_2, b_1.$ $b \leftarrow a_1.$ $b \leftarrow a_2, \neg b_1.$ $b \leftarrow \neg a_2, \neg b_2.$ $b \leftarrow \neg b_2, \neg b_1.$ $b \leftarrow \neg b_2, a_1.$ | $b \leftarrow a_1.$ (ϵ, ϵ) $b \leftarrow a_1.$ |
| $ab \rightarrow \epsilon \rightarrow \epsilon$ | | $a \rightarrow \epsilon \rightarrow b$ | | $\epsilon \rightarrow b \rightarrow \epsilon$ | |
| $(a_2 b_2, \epsilon)$ | $(a_1 b_1, \epsilon)$ | (a_2, ϵ) | (a_1, ϵ) | (b_1, ϵ) | (ϵ, b) |
| $a \leftarrow \neg a_2, b_1.$ $a \leftarrow a_2, b_2, a_1.$ $a \leftarrow a_2, b_2, \neg b_1.$ $a \leftarrow b_2, b_1.$ $a \leftarrow a_1.$ $a \leftarrow a_2, \neg b_2, \neg b_1.$ $a \leftarrow \neg a_2, a_1, \neg b_1.$ $b \leftarrow a_2, \neg b_2.$ $b \leftarrow \neg b_2, b_1.$ $b \leftarrow a_1.$ $b \leftarrow \neg a_2, \neg b_2, \neg b_1.$ $b \leftarrow \neg a_2, a_1, \neg b_1.$ | $b \leftarrow a_1, \neg b_1.$ (ϵ, ϵ) $b \leftarrow a_1, \neg b_1.$ | $a \leftarrow \neg a_2, b_1.$ $a \leftarrow b_2, b_1.$ $a \leftarrow a_1.$ $a \leftarrow \neg a_2, \neg b_2, \neg b_1.$ $b \leftarrow a_2, \neg b_2.$ $b \leftarrow \neg b_2, b_1.$ $b \leftarrow a_1.$ | $b \leftarrow \neg a_1, \neg b_1.$ (ϵ, b) \emptyset | $a \leftarrow \neg a_2, \neg b_2, \neg b_1.$ $a \leftarrow \neg a_2, a_1, \neg b_1.$ $a \leftarrow b_2, b_1.$ $a \leftarrow a_1.$ $b \leftarrow a_2, \neg b_2.$ $b \leftarrow \neg a_2, \neg b_2, \neg b_1.$ $b \leftarrow \neg a_2, a_1, \neg b_1.$ $b \leftarrow a_1.$ | \emptyset (b_1, ϵ) \emptyset |
| Merging of the programs | | OUTPUT | | | |
| $a \leftarrow b_2, b_1.$ $a \leftarrow a_1.$ $b \leftarrow a_2, \neg b_2.$ $b \leftarrow a_1.$ | | $a \leftarrow b_2, b_1.$ $b \leftarrow a_2, \neg b_2.$ | | | |

TABLE 4.1: Execution of **LFkT** on traces of figure 4.4

Here the algorithm learns 2-step and 1-step relations that are represented by two different sets of rules. Both of them are initialized with the most general hypotheses, i.e. rules with empty body. The algorithm then analyzes the first trace $a \rightarrow b \rightarrow b$. From this trace, it extracts a 2-step

interpretation transition: $(a_{t-2}b_{t-1}, b)$ (abridged (a_2b_1, b) in the table to save space); and two 1-step interpretation transitions: (a_{t-1}, b) and (b_{t-1}, b) that respectively represent the begin and the end of the trace. The aforementioned 2-step interpretation is used to revise the 2-step rules and the 1-step interpretations are used to revise the 1-step rules. From $(a_{t-2}b_{t-1}, b)$, since a is not present in the second interpretation, the algorithm infers the anti-rule $R := a \leftarrow a_{t-2}, b_{t-1}$. R is used to revise the 2-step rules that subsume it, so that these rules become consistent with the interpretation. This is done by the addition of the negation of each element of the body of R into the rules a_t and b_t . From (a_{t-1}, b) it also infers an anti-rule of a that is $R' := a \leftarrow a_{t-1}$. The 1-step rules are revised by R' and then it analyzes the second 1-step interpretation transition. From (b_{t-1}, b) , it infers the anti-rule $R'' := a \leftarrow b_{t-1}$, that is now used to revise the new 1-step rules. Here the rule of b is not modified because it is consistent with the interpretation and, by extension, consistent with the trace.

Now, **LFkT** analyzes the trace $ab \rightarrow b \rightarrow a$. The 2-step rules of b are revised by the anti-rule $R := b \leftarrow a_{t-2}, b_{t-2}, b_{t-1}$ extracted from $(a_{t-2}b_{t-2}b_{t-1}, a)$. The 1-step rules are revised by $R' := a \leftarrow a_{t-1}, b_{t-1}$ extracted from $(a_{t-1}b_{t-1}, b)$. Here, the rule $a \leftarrow a_{t-1}, b_{t-1}$ is removed because it subsumes R' (here it is R' itself) and cannot be specified. This kind of deletion is represented by double stroked rules. the rule of b is then revised by the second 1-step interpretation.

Then, the algorithm analyzes the trace $b \rightarrow b \rightarrow a$. Some 2-step rules of b are specialized to become consistent with the trace. Some of them are removed because after specialization they are subsumed by other rules and then become useless. These rules are stroked in the table, it is the case of $R := b \leftarrow \neg a_{t-2}, \neg b_{t-2}$ the specialization of $R' := b \leftarrow \neg a_{t-2}$ by adding b_{t-2} . This rule is subsumed by $R'' := b \leftarrow \neg b_{t-2}$ and since R'' is consistent with the trace, we do not need to keep R because all rules that can be specialized from R can also be obtained by R'' .

Solving continues until all traces have been analyzed. At the end, the 2-step rules are merged with the 1-step rules by removing the 1-step rules from the set of 2-step rules. These rules can be non consistent with some 1-step interpretations since they have not been revised by all of them. They can safely be removed, because if they are consistent with all 1-step interpretations, this means they will appear in the set of 1-step rules.

LFkT finally outputs the two rules $R_1 := a \leftarrow b_{t-2}, b_{t-1}$ and $R_2 := b \leftarrow a_{t-2}, \neg b_{t-2}$, that correspond to the system of Example 4.1. To ensure that the output is correct it will need to analyzes all possible traces of the system that is 2^{n*k} , with n the number of variables of the system and k the size of a trace. Here it will require the analysis of all 16 possible traces. But, in this example, the algorithm succeeds to learn the rules of the system after the analysis of 8 traces of execution.

| Run time (# of seconds) | | | | | | Output size (# of rules) | | | | | |
|-------------------------|-------------|--------|---------------|--------|--------|--------------------------|-----|-----|------------|-------|-------|
| # Traces | k=1 | k=2 | k=3 | k=4 | k=5 | # Traces | k=1 | k=2 | k=3 | k=4 | k=5 |
| 10 | 0.23s | 0.27s | 0.16s | 0.28s | 0.31s | 10 | 994 | 735 | 413 | 1,162 | 1,278 |
| 100 | 1.87s | 2.49s | 1.63s | 2.37s | 2.84s | 100 | 837 | 736 | 383 | 1,025 | 1,156 |
| 1,000 | 15.3s | 18.5s | 13.3s | 20.1s | 23.8s | 1,000 | 835 | 647 | 364 | 942 | 1,096 |
| 10,000 | 146s | 218s | 147s | 201s | 234s | 10,000 | 728 | 739 | 408 | 857 | 987 |
| 100,000 | 2,177s | 2,577s | 1,643s | 1,764s | 2,243s | 100,000 | 929 | 535 | 390 | 788 | 950 |
| 1,000,000 | 27,768s | 22,517 | 12,384 | 15,670 | 20,413 | 1,000,000 | 833 | 559 | 350 | 769 | 930 |

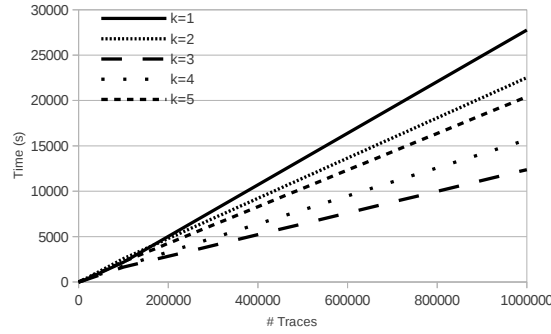


FIGURE 4.2: LFKT Run time varying the input size (number of traces)

4.1.4 Evaluation

In the previous subsections, we have illustrated step by step how LFKT algorithm is able to learn Markov(k) systems. To illustrate the merits of our work, we now apply this approach to the analysis of the yeast cell cycle dataset from [89] and [90], which have been previously analyzed in [91]. In this paper, Li et al. tackle the inference of gene regulatory networks from temporal gene expression data. The originality of their work lies in the fact they consider delayed correlations between genes. The methodology can capture gene regulations that are delayed of k time units. The limits of the approach is that the authors only consider pairwise overlaps of expression levels shifted in time relative to each other. Another limit of the approach is that it is not able to make a distinction between a causal gene-gene regulation and the scenarios where two genes, A and B, are being co-regulated by a third gene C: do we have A that regulates B that regulates C, or is it a cooperation between A and B that regulates C?

Here, starting from a set of different traces coming from the yeast cell cycle system, we have performed various experiments where we have tuned the number of traces that have been considered on the one hand, the value of k (i.e., the number of time steps representing the memory of the system) on the other hand.

Figure 4.5 shows the evolution of run time of learning with LFKT on the five Boolean networks of the yeast cell cycle proposed by [91]. These five programs are respectively Markov(1) to Markov(5). In these experiments, for each Boolean network the number of variable is 16 and the length of traces in input is five states. The five Boolean networks have been implemented as a logic program using Answer Set Programming [92]. The source code of these programs are given as supplementary material. Traces of executions of these programs have been computed

using the answer set solver `clasp` [86]. All experiments are run with a C++ implementation of **LFkT** on a processor Intel Xeon (X5650, 2.67GHz) with 12GB of RAM. The main purpose of these experiments is to assess the efficiency of our approach, i.e., how many traces **LFkT** can handle for a given k . Complete output of LFkT for these experiments is accessible as textfile at <http://tony.research.free.fr/paper/Frontier/output.zip>.

In the first table of Figure 4.5, the evolution of run time from 10 to 1,000,000 traces (which is arbitrary chosen as upper bound of the scalability of the experiments) shows that, in practice, learning with **LFkT** is linear in the number of traces when the number of variables is fixed. Results show that the algorithm can handle more than one million of traces in less than 10 hours. Since each trace is a sequence of five state transitions, when learning the Markov(5) system, each trace can be decomposed into 15 interpretation transitions (one 5-step, two 4-step, three 3-step, four 2-step and five 1-step). Learning the Markov(5) program from one million traces of executions of size five requires the processing of 15 million of interpretation transitions. Learning the Markov(4) to Markov(1) programs requires to process respectively 14 million, 12 million, 9 million and 5 million of interpretation transitions. Intuitively one could expect that learning the Markov(2) system to take significantly more time than learning the Markov(1) system. But each program is different, i.e., the Markov(2) program is not an extension of the Markov(1) program with 2-step rules. That is why run time is not always larger for a larger k : learning time also depends of the rules that are learned. In this experiment, the best run time is obtained with the Markov(3) program. We cannot say that the rules of this program are simpler than the others, but they are simpler to learn for the algorithm. In the second table, we observe that the number of rules learned for the Markov(3) program is significantly smaller than for the others. It means that the algorithm needs to compare less rules for each traces analysis, which can explain the speed up.

In this benchmark, in order to be faithful to the biological experiments presented by [91], we considered $k = 5$ as a maximum. But our algorithm succeeds in processing larger memory effects. On some random dummy examples (accessible at the above mentioned URL), we were able to learn Markov(7) systems with the following performances: we can learn 10 traces in 2.8s, 100 traces in 27s, 1,000 traces in 249s, 10,000 traces in 3,621s, 100,000 traces in 39,973s, 1,000,000 traces in 441,270s. Even if the computation time increases, it should be kept in mind that our method is designed to allow successive refinements of a model about its memory effect. These results show that such an approach is tractable even with a large number of input traces.

We now apply **LFkT** to learn our previous benchmarks. These Boolean networks are taken from Dubrova and Teslenko [77]: these networks model the control of flower morphogenesis in *Arabidopsis thaliana*, the budding yeast cell cycle regulation, the fission yeast cell cycle regulation and the mammalian cell cycle regulation.

4.1.5 Conclusion

To understand the memory effect involved in some interactions between biological components, it is necessary to include delayed influences in the model. In this section, we proposed a logical method to learn such models from state transitions systems. We designed an approach to learn Boolean networks with delayed influences. This section introduced our approach to learn normal logic programs from interpretation transitions on k -steps. This can be directly applied to the learning of Boolean 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 [91]. 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. In account of the noise inherent to biological data, the ability to either perform an efficient discretization of the data or to include the notion of noise inside the modeling framework is fundamental. We will thus have to discuss the discretization procedure and the robustness of our modeling against noisy data and compare it to existing approaches, like the Bayesian ones [93].

4.2 Multivalued Variables

In this section, we provide the formal extension of *LF1T* to multivalued variable. By considering variables with multiples values we can represent and learn more expressive models than Boolean ones. The precise description of interactions between biological components generally requires more than Boolean values (e.g., when one component influences two others, it is very unlikely that the influences are triggered by exactly the same concentration of the component). Research in multi-valued logic programming has proceed along three different directions [94]: bilattice-based logics [95, 96], quantitative rule sets [97] and annotated logics [98, 99]. The multi-valued logic representation used in our new algorithm is based on annotated logics. Here, to each atom corresponds a given set of values. In a rule, a literal is an atom annotated with one of these values. It allows us to represent annotated atoms simply as classical atoms and thus to remain in the normal logic program semantics.

4.2.1 Formalization

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. 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

$$var^{val} \leftarrow var_1^{val_1} \wedge \dots \wedge var_n^{val_n} \quad (4.1)$$

where var^{val} and $var_i^{val_i}$'s are atoms ($n \geq 1$). Like before, for any rule R of the form (4.1), left part of \leftarrow 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 (4.1) as $b(R) = \{var_1^{val_1}, \dots, var_n^{val_n}\}$. A rule R of the form (4.1) is interpreted as follows: the variable var takes the value val in the next state if all variables var_i have the value val_i in the current state.

Example 4.4. Let consider the following rules, $R_1 = a^1 \leftarrow b^1$, $R_2 = b^1 \leftarrow a^1 \wedge b^0$, $R_3 = a^0 \leftarrow b^0$, $R_4 = a^0 \leftarrow b^0$, $R_5 = b^0 \leftarrow a^0$, $R_6 = b^0 \leftarrow b^1$. The logic program $P = \{R_1, R_2, R_3, R_4, R_5, R_6\}$ is a multi-valued logic program.

An interpretation of a multi-valued program provides the value of each variable of the system and is defined as follows.

Definition 4.9 (Multi-valued Interpretation). Let \mathcal{B} be a set of atoms where each element has the form var^{val} . An *interpretation* I of a set of atoms \mathcal{B} is a subset of \mathcal{B} where $\forall var^{val} \in \mathcal{B}, var^{val'} \in I$ and $\forall var^{val'} \in I, \nexists var^{val''} \in I, val' \neq val''$.

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 interpretations $(I, next(I))$.

Definition 4.10 (Multi-valued Model). An interpretation I is a model of a program P if $b(R) \subseteq I$ implies $h(R) \in I$ for every rule R in P .

Definition 4.11 (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 .

Definition 4.12 (Subsumption). Let R_1 and R_2 be two rules. If $h(R_1) = h(R_2)$ and $b(R_1) \subseteq b(R_2)$ then R_1 subsumes R_2 . Let P be a logic program and R be a rule. P subsumes R if there exists a rule $R' \in P$ that subsumes R .

We say that a rule R_1 is *more general* than another rule R_2 if R_1 subsumes R_2 . In particular, a rule R is *most general* if there is no rule $R' (\neq R)$ that subsumes R ($b(r) = \emptyset$).

Example 4.5. Let R_1 and R_2 be the two following rules: $R_1 = (a^1 \leftarrow b^1)$, $R_2 = (a^1 \leftarrow a^0 \wedge b^1)$, R_1 subsumes R_2 because $(b(R_1) = \{b^1\}) \subset (b(R_2) = \{a^0, b^1\})$. When R_1 appears in a logic program P , R_2 is useless for P , because whenever R_2 can be applied, R_1 can be applied.

To learn multi-valued logic programs with **LF1T** we need to adapt the ground resolution and the least specialization to handle non-boolean variables.

Definition 4.13 (complement). Let R_1 and R_2 be two rules, R_2 is a complement of R_1 on var^{val} if $var^{val} \in b(R_1)$, $var^{val'} \in b(R_2)$, $val \neq val'$ and $(b(R_2) \setminus \{var^{val'}\}) \subseteq (b(R_1) \setminus \{var^{val}\})$.

Definition 4.14 (multi-valued ground resolution). Let R be a rule, P be a logic program and \mathcal{B} be a set of atoms, R can be generalized on var^{val} if $\forall var^{val'} \in \mathcal{B}, val \neq val', \exists R' \in P$ such that R' is a complement of R on var^{val} :

$$generalise(R, P) = h(R) \leftarrow b(R) \setminus var^{val}$$

Definition 4.15 (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

$$ls(R_1, R_2, \mathcal{B}) = \{h(R_1) \leftarrow b(R_1) \wedge var^{val'} \mid var^{val} \in b(R_2) \setminus b(R_1), var^{val'} \in \mathcal{B}, val' \neq val\}$$

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:

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

4.2.2 Algorithm

Now we present the adaptation of the **LF1T** algorithm to learn multi-valued logic program. The algorithm guarantees that the output only contains the minimal rules that realize the input transitions. Algorithm 15 shows the pseudo-code of the new **LF1T**.

Algorithm 15 **LF1T**(E, \mathcal{B}) : Learn a program P that realize E

```

1: INPUT:  $E$  a set of state transitions of a system  $S$  and  $\mathcal{B}$  the set of all possible atoms that can appear
   in  $I$  and  $J$ .
2: OUTPUT: An logic program  $P$  such that  $J = next(I)$  holds for any  $(I, J) \in E$ .

3:  $P := \emptyset$ 
   // Initialize  $P$  with the most general rules
4: for each  $var^{val} \in \mathcal{B}$  do
5:    $P := P \cup \{var^{val} \leftarrow .\}$ 
6: end for
   // Specify  $P$  to realize each state transition
7: while  $E \neq \emptyset$  do
8:   Pick  $(I, J) \in E$ ;  $E := E \setminus \{(I, J)\}$ 
9:   for each atom  $var^{val} \in J$  do
10:    for each  $var^{val'} \in \mathcal{B}, val' \neq val$  do
11:       $R_{var^{val'}}^I := var^{val'} \leftarrow \bigwedge_{l_i \in I} l_i$ 
12:       $P := \text{Specialize}(P, R_{var^{val'}}^I, \mathcal{B})$ 
13:    end for
14:   end for
15: end while
16: return  $P$ 

```

Like in previous versions, **LF1T** takes a set of state transitions E as input and outputs a logic program P that realizes E . To guarantee the minimality of the learned NLP, **LF1T** starts with the most general rules (lines 3-7). The idea is that, at the begining we consider that each variable can take any of its values with no conditions: for all state, the next state can be any state.

Then **LF1T** iteratively analyzes each transition $(I, J) \in E$ (lines 8-16). For each atoms var^{val} that **does not appear** in J , **LF1T** infers an **anti-rule** $R_{var^{val}}^I$ (lines 11-12):

$$R_{var^{val}}^I := var^{val} \leftarrow \bigwedge_{B_i \in I} B_i$$

. Then, **LF1T** uses least specialization to make P consistent with all $R_{var^{val}}^I$ (line 13). The idea here, is to specialize all rules which state that the variable var should take another value val than the one observed in J . Algorithm 27 shows in detail the pseudo code of this operation.

Algorithm 16 $\text{specialize}(P, R, \mathcal{B})$: specialize P to avoid the subsumption of R

```

1: INPUT: a logic program  $P$  and a rule  $R$ 
2: OUTPUT: the least specialization of  $P$  by  $R$ .

3:  $conflicts$  : a set of rules
4:  $conflicts := \emptyset$ 
   // Search rules that need to be specialized
5: for each rule  $R_P \in P$  do
6:   if  $R_P$  subsumes  $R$  then
7:      $conflicts := conflicts \cup R_P$ 
8:      $P := P \setminus R_P$ 
9:   end if
10: end for
   // Revise the rules by least specialization
11: for each rule  $R_c \in conflicts$  do
12:   for each literal  $v^{val} \in b(R)$  do
13:     if  $v^{val} \notin b(R_c)$  then
14:       for each  $v^{val'} \in \mathcal{B}, val' \neq val$  do
15:          $R'_c := (h(R_c) \leftarrow (b(R_c) \cup v^{val'}))$ 
16:         if  $P$  does not subsume  $R'_c$  then
17:            $P := P \setminus \text{all rules subsumed by } R'_c$ 
18:            $P := P \cup R'_c$ 
19:         end if
20:       end for
21:     end if
22:   end for
23: end for
24: return  $P$ 

```

LF1T first extracts all rules $R_P \in P$ that subsume $R_{var^{val}}^I$ (lines 3-10). It generates the least specialization of each R_P by generating a rule for each literal in $R_{var^{val}}^I$. Each rule contains all literals of R_P plus a new literal. This literal gives another value to the variable of a literal of $R_{var^{val}}^I$, so that $R_{var^{val}}^I$ is not subsumed by that rule. Then **LF1T** adds in P all the generated rules that are not subsumed by P (line 15-17), so that P becomes consistent with the transition (I, J) and all rules of P remains minimal. When all transitions have been analyzed, **LF1T** outputs P that has become a list of minimal rule that realizes E .

4.3 Multivalued Delayed Systems

In a previous section we presented an algorithm to learn timed Boolean networks. 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 section, 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.

4.3.1 Formalization

Definition 4.16 (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 4.16, 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 4.17 (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. Example 4.6 shows the Boolean Markov(2) system of Example 4 of [73] represented as a multi-valued logic program. Figure 4.3 shows the interaction graph of this system and the corresponding transition diagram.

Example 4.6. Let consider the following rules, $R_1 = a^1 \leftarrow b_{t-1}^1 \wedge b_{t-2}^1$, $R_2 = b^1 \leftarrow a_{t-2}^1 \wedge b_{t-2}^0$, $R_3 = a^0 \leftarrow b_{t-2}^0$, $R_4 = a^0 \leftarrow b_{t-1}^0$, $R_5 = b^0 \leftarrow a_{t-2}^0$, $R_6 = b^0 \leftarrow b_{t-2}^1$. The logic program $S = \{R_1, R_2, R_3, R_4, R_5, R_6\}$ is a Markov(2) system, i.e., the state of the system depends on the two previous states. The value of a is 1 at time step t only if the value of b was 1 at $t - 1$

and $t - 2$. The value of b is 1 at time step t only if the value of a was 1 at $t - 2$ and the value of b was 0 at $t - 2$. The atoms that appear in the head of the rules of S are $\{a^0, a^1, b^0, b^1\}$. B_1 represents these atoms from time step $t - 1$: $B_1 = \{a_{t-1}^0, a_{t-1}^1, b_{t-1}^0, b_{t-1}^1\}$ and B_2 represents these atoms from time step $t - 2$: $B_2 = \{a_{t-1}^0, a_{t-1}^1, b_{t-1}^0, b_{t-1}^1, a_{t-2}^0, a_{t-2}^1, b_{t-2}^0, b_{t-2}^1\}$. Here S is the Markov(2) system that produces the transitions of figure 4.4. The interaction graph of S is shown by figure 4.3.

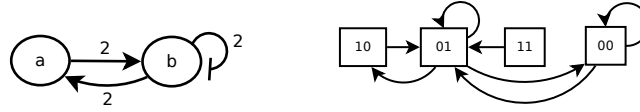


FIGURE 4.3: The interaction graph of the Markov(2) system of Example 4.6 (left) and its state transitions diagram (right). Here, activations and inhibitions are labeled by the delay of the influence.

Trace of execution, their consistency and k -step interpretation are formally equivalent to the Boolean case. Except that state are multi-valued interpretation (each variable value is given explicitly).

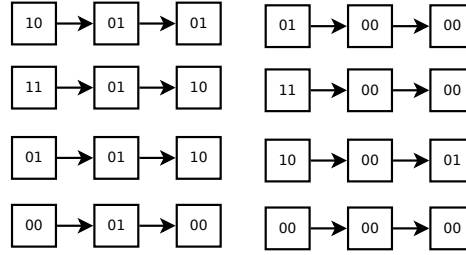


FIGURE 4.4: Eight traces of executions of the system of Example 4.6

Example 4.7. The trace $11 \rightarrow 01 \rightarrow 10$ can be interpreted in the three following ways:

- $(a_{t-2}^1 b_{t-2}^1 a_{t-1}^0 b_{t-1}^1, a^1 b^0)$: the 2-step interpretation transition that corresponds to the full trace $11 \rightarrow 01 \rightarrow 10$.
- $(a_{t-1}^1 b_{t-1}^1, a^0 b^1)$: the 1-step interpretation transition corresponding to the sub-trace $11 \rightarrow 01$.
- $(a_{t-1}^0 b_{t-1}^1, a^1 b^0)$: the 1-step interpretation transition that corresponds to the sub-trace $01 \rightarrow 10$.

4.3.2 Algorithm

In section 4.1 we proposed a method to learn delayed influences of Boolean systems: the *LFkT* algorithm. In this section, we propose a new version of this algorithm that handle multi-valued variables. Furthermore, the delays are now computed dynamically and does not need to be known or fixed to the maximal value (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 17. Like before, the idea of the algorithm is to start with the most general rules and use least specialization iteratively on each traces to make the rules learned consistent with the all input observations.

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 18. If this delay is greater than the one currently considered by the algorithm, it updates this delay and generates programs 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 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 $\exists R \in P'_k, b(R) \subseteq I$ 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 **interpret**, whose pseudo code is given in Algorithm 20. Here, $\min(k, |T|)$

Algorithm 17 LFkT(O, \mathcal{B}) : Learn a set of rules that realize O

```

1: INPUT:  $O$  a set of traces of executions,  $\mathcal{B}$  a set of atoms
2: OUTPUT:  $P$  a logic program that realizes the transitions of  $O$ .
3:  $P'$  a vector of set of rules
4:  $E$  a set of pairs of interpretations  $(I, J)$ 
5:  $k$  an integer
6: // 1) Initialize  $P'$  with the most general logic program
7: for each atom  $var^{val} \in \mathcal{B}$  do
8:    $P'_1 := P'_1 \cup \{var^{val} \leftarrow\}$ 
9: end for
10:  $k := 1$  // Assume Markov(1)
11: // 2) Learning phase
12: while  $O \neq \emptyset$  do
13:   pick a trace  $T \in O$ 
14:   // 2.1) Check delay of the trace
15:   if  $\text{delay}(T) > k$  then // Extend the delay to learn
16:     for  $i = k + 1$  to  $\text{delay}(T)$  do
17:       for each atom  $var^{val} \in \mathcal{B}$  do
18:         for each atom  $var'^{val'} \in \mathcal{B}$  do
19:            $P'_i := P'_i \cup \{var^{val} \leftarrow var'^{val'}_{t-i}\}$ 
20:         end for
21:       end for
22:     end for  $k := \text{delay}(T)$ 
23:     for each trace  $T' \in O'$  do
24:        $P' := \text{learn}(P, T', k, \mathcal{B})$ 
25:     end for
26:   end if
27:   // 2.2) Check consistency with previous traces
28:   if  $\exists T' \in O', T$  and  $T'$  are not  $k$ -consistent then
29:     for each  $k'$  from  $k$  to  $\min(|T|, |T'|)$  do
30:       if  $T$  and  $T'$  are  $k'$ -consistent then
31:         for  $i = k$  to  $k'$  do
32:           for each atom  $var^{val} \in \mathcal{B}$  do
33:             for each atom  $var'^{val'} \in \mathcal{B}$  do
34:                $P'_i := P'_i \cup \{var^{val} \leftarrow var'^{val'}_{t-i}\}$ 
35:             end for
36:           end for
37:         end for
38:       for each trace  $T' \in O'$  do
39:          $P' := \text{learn}(P, T', k, \mathcal{B})$ 
40:       end for
41:        $k := k'$ 
42:     else //  $T$  and  $T'$  are not consistent, cannot happen if  $O$  is consistent
43:       EXIT: non-deterministic input
44:     end if
45:   end for
46: end if
47: // 2.3) Specify  $P'$  by the interpretations of the trace
48:  $P' := \text{learn}(P, T, 1, \mathcal{B})$ 
49:  $O := O \setminus \{T\}$ 
50:  $O' := O' \cup \{T\}$ 
51: end while
52: // 3) Merge the programs into a unique logic program
53:  $\text{merging} := \emptyset$ 
54: for each  $i$  from 1 to  $k$  do
55:   remove from  $P'_i$  all rules subsumed by a rule of  $\text{merging}$ 
56:    $\text{merging} := \text{merging} \cup P'_i$ 
57: end for
58: // 4) Keep only the rules that can realize the observations
59:  $P := \emptyset$ 
60: for each  $T' \in O'$  do
61:    $E := \text{interpret}(T')$ 
62:   for each  $(I, J) \in E$  do
63:     for each  $R \in \text{merging}$  do
64:       if  $b(R) \subseteq I$  and  $h(R) \in J$  then
65:          $P := P \cup \{R\}$ 
66:       end if
67:     end for
68:   end for
69: end for
70: return  $P$ 

```

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, \dots , one 1-step interpretation. The function

Algorithm 18 $\text{delay}(T)$: Compute the minimal delay of a trace

```

1: INPUT: a trace of execution  $T = (S_0, \dots, S_n)$ 
2: OUTPUT:  $\text{delay}$  an integer

3:  $\text{delay} := 1$ 
4: for each  $i$  from 1 to  $n - 1$  do
5:   for each  $j$  from  $i$  to  $n - 1$  do
6:     if  $S_i = S_j$  then
7:        $k := 1$ 
8:       while  $k \leq i$  AND  $S_{i-k} = S_{j-k}$  do
9:          $k := k + 1$ 
10:      end while
11:       $\text{delay} := \max(\text{delay}, k)$ 
12:    end if
13:  end for
14: end for
15: return  $\text{delay}$ 

```

Algorithm 19 $\text{learn}(P, T, \text{min_delay}, \mathcal{B})$: Revise P to avoid the subsumption of R

```

1: INPUT:  $P$  a vector of logic program,  $T$  a trace of execution and  $\text{min\_delay}$  an integer
2: OUTPUT: a vector of logic program

3:  $E := \text{interpret}(T)$ 
4: for each  $i$  from  $\text{min\_delay}$  to  $|T|$  do
5:   for each  $k$ -step interpretation  $(I, J) \in E$  with  $k \geq i$  do
6:     remove from  $I$  all atoms  $\text{var}_{t-n}^{\text{val}}$  with  $n > i$ 
7:     for each atom  $\text{var}^{\text{val}} \in J$  do
8:       for each  $\text{var}^{\text{val}'} \in \mathcal{B}, \text{val}' \neq \text{val}$  do
9:          $R_{\text{var}^{\text{val}'}}^I := \text{var}^{\text{val}'} \leftarrow \bigwedge_{l_j \in I} l_j$ 
10:         $P_i := \text{Specialize}(P_i, R_{\text{var}^{\text{val}'}}^I, \mathcal{B})$ 
11:      end for
12:    end for
13:  end for
14: end for
15: return  $P$ 

```

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 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**: $R_{\text{var}^{\text{val}}}^I := \text{var}^{\text{val}} \leftarrow \bigwedge_{B_i \in I} B_i$. Then, least specialization is used to make each corresponding logic program P'_i consistent with $R_{\text{var}^{\text{val}}}^I$, according to the delay of interpretation transition. Algorithm 21 shows the pseudo code of this operation. In the function **specialize**, it first extracts all rules $R_P \in P$ that subsumes R_A^I . It generates the least specialization of each R_P by generating a rule for each literal in $R_{\text{var}^{\text{val}}}^I$. Each rule contain all literals of R_P , plus a literal that represents another value of the variable represented by a literal in $R_{\text{var}^{\text{val}}}^I$, so that

Algorithm 20 $\text{interpret}(T)$: Extract interpretation transitions from a trace

```

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

3:  $E := \emptyset$ 
   // Extract interpretations
4: for each  $k$  from 1 to  $|T|$  do
5:    $T' := (S_0, \dots, S_k)$  // the sub-trace of size  $k$  of  $T$  that start from  $S_0$ 
6:    $I := \emptyset$ 
7:   for each state  $s_{k'}$  before  $s_k$  in  $T'$  do
8:      $\text{delay} := k - k'$ 
9:     for each atom  $a \in s_{k'}$  do
10:       $I := I \cup \{a_{t-\text{delay}}\}$ 
11:     end for
12:    $E := E \cup (I, S_k)$ 
13: end for
14: end for
15: return  $E$ 

```

Algorithm 21 $\text{specialize}(P, R, \mathcal{B})$: specialize P to avoid the subsumption of R

```

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

3:  $\text{conflicts}$  : a set of rules
4:  $\text{conflicts} := \emptyset$ 
   // Search rules that need to be specialized
5: for each rule  $R_P \in P$  do
6:   if  $R_P$  subsumes  $R$  then
7:      $\text{conflicts} := \text{conflicts} \cup R_P$ 
8:      $P := P \setminus R_P$ 
9:   end if
10: end for
   // Revise the rules by least specialization
11: for each rule  $R_c \in \text{conflicts}$  do
12:   for each literal  $\text{var}_{t-k}^{\text{val}} \in b(R)$  do
13:     if  $\text{var}_{t-k}^{\text{val}} \notin b(R_c)$  then
14:       for each  $\text{var}_{t-k}^{\text{val}'} \in \mathcal{B}, \text{val}' \neq \text{val}$  do
15:          $R'_c := (h(R_c) \leftarrow (b(R_c) \cup \text{var}_{t-k}^{\text{val}'}))$ 
16:         if  $P$  does not subsume  $R'_c$  then
17:            $P := P \setminus \text{all rules subsumed by } R'_c$ 
18:            $P := P \cup R'_c$ 
19:         end if
20:       end for
21:     end if
22:   end for
23: end for
24: return  $P$ 

```

$R_{var^{val}}^I$ 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 4.18 (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. Let V be the vector of interpretation transition extracted from O by **LFkT** (Algorithm 20). According to Theorem 4 of [64], initializing *LFIT* 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 [64], 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 $\forall (I, J) \in v_n, \exists (I, J'), J \neq J'$.

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 . □ □

Theorem 4.19 (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. n is the number of possible headsof 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 $O(n \cdot v^{nk+1} + |O|)$.

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_A^I for each $A \in \mathcal{B}$, $A \notin J$. **LFkT** compares each R_A^I with all rules of each programs P_i . There is at most $|\mathcal{B}| - n$ anti-rules that can be inferred 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 $O(|T| \cdot |\mathcal{B}| - n \cdot k |P_i|) = O(|T| \cdot nv - n \cdot k \cdot \frac{n \cdot v^{nk}}{k}) = O(|T| \cdot n^2 v^{nk+2} - n)$ 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 $O(|T|^2) = (n^2 k^2)$. And, checking the consistency of a new traces with previous analyzed one is bound by $O(|O| * n^2 k^2)$. 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 rules with all k -step interpretation 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 $O(\sum_{T \in O} |T| \cdot nv^{nk+3} + |O| \cdot n^2 k^2 + n \cdot v^{nk+2})$. □ □

4.3.3 Running example of LFkT

Table 4.2 shows the execution of **LFkT** on 4 traces of figure 4.4 where $(a_{t-2}^1 b_{t-2}^1 a_{t-1}^0 b_{t-1}^1, a^1 b^0)$ represents the interpretation of the trace $11 \rightarrow 01 \rightarrow 10$. Introduction of a literal by least specialization is represented in bold and rules that are subsumed after specialization are stroked.

| Trace | Initialization | $a^1 b^0 \rightarrow a^0 b^1 \rightarrow a^0 b^1$ | | $a^1 b^1 \rightarrow a^0 b^1 \rightarrow a^1 b^0$ | | Check $O' = \{a^1 b^0 \rightarrow a^0 b^1 \rightarrow a^0 b^1\}$ | |
|----------------|--------------------------------------|--|--|--|--|--|--------|
| Interpretation | | $(a_{t-2}^1 b_{t-2}^0 a_{t-1}^0 b_{t-1}^1, a^0 b^1)$ | $(a_{t-1}^1 b_{t-1}^0, a^0 b^1)$ | Conflict with $a^1 b^0 \rightarrow a^0 b^1 \rightarrow a^0 b^1$ | | $(a_{t-2}^1 b_{t-2}^0 a_{t-1}^0 b_{t-1}^1, a^0 b^1)$ | |
| Program | P'_1 | P'_1 | P'_1 | P'_1 | P'_2 | P'_1 | P'_2 |
| Rules | $a^0.$ $a^1.$ $b^0.$ $b^1.$ | $a^0.$ $a^1 \leftarrow a_{t-1}^1.$ $a^1 \leftarrow b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1.$ $b^0 \leftarrow b_{t-1}^0.$ $b^1.$ | $a^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1.$ | $a^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1.$ | $a^0 \leftarrow a_{t-2}^1.$ $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^0 \leftarrow b_{t-2}^1.$ $a^1 \leftarrow a_{t-2}^0.$ $a^1 \leftarrow a_{t-1}^0.$ $a^1 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow b_{t-2}^1.$ $b^0 \leftarrow a_{t-1}^1.$ $b^0 \leftarrow a_{t-2}^0.$ $b^0 \leftarrow b_{t-1}^0.$ $b^0 \leftarrow b_{t-2}^1.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow a_{t-2}^0.$ $b^1 \leftarrow b_{t-1}^0.$ $b^1 \leftarrow b_{t-2}^1.$ | $a^0 \leftarrow a_{t-2}^1.$ $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^0 \leftarrow b_{t-2}^1.$ $a^1 \leftarrow a_{t-2}^0.$ $a^1 \leftarrow a_{t-1}^0.$ $a^1 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow b_{t-2}^1.$ $b^0 \leftarrow a_{t-1}^1.$ $b^0 \leftarrow a_{t-2}^0.$ $b^0 \leftarrow b_{t-1}^0.$ $b^0 \leftarrow b_{t-2}^1.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow a_{t-2}^0.$ $b^1 \leftarrow b_{t-1}^0.$ $b^1 \leftarrow b_{t-2}^1.$ | |

| $a^1 b^1 \rightarrow a^0 b^1 \rightarrow a^1 b^0$ | | | $a^1 b^1 \rightarrow a^0 b^0 \rightarrow a^0 b^0$ | | | $a^0 b^0 \rightarrow a^0 b^1 \rightarrow a^0 b^0$ | | |
|--|--|--|--|--|--|--|--|--|
| $(a_{t-2}^1 b_{t-2}^1 a_{t-1}^0 b_{t-1}^1, a^1 b^0)$ | | | $(a_{t-2}^1 b_{t-2}^1 a_{t-1}^1 b_{t-1}^0, a^0 b^0)$ | | | $(a_{t-1}^0 b_{t-1}^0 a_{t-1}^1 b_{t-1}^1, a^0 b^0)$ | | |
| P'_1 | P'_2 | P'_1 | P'_1 | P'_2 | P'_1 | P'_1 | P'_2 | P'_1 |
| $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ | $a^0 \leftarrow a_{t-2}^1.$ $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^0 \leftarrow b_{t-2}^1.$ $a^1 \leftarrow a_{t-2}^0.$ $a^1 \leftarrow a_{t-1}^0.$ $a^1 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow b_{t-2}^1.$ $b^0 \leftarrow a_{t-1}^1.$ $b^0 \leftarrow a_{t-2}^0.$ $b^0 \leftarrow b_{t-1}^0.$ $b^0 \leftarrow b_{t-2}^1.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow a_{t-2}^0.$ $b^1 \leftarrow b_{t-1}^0.$ $b^1 \leftarrow b_{t-2}^1.$ | $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ | $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ | $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ | $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ | $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ | $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ | $a^0 \leftarrow a_{t-1}^1.$ $a^0 \leftarrow b_{t-1}^0.$ $a^1 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $a^1 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^0 \leftarrow a_{t-1}^1, b_{t-1}^1.$ $b^0 \leftarrow a_{t-1}^0, b_{t-1}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow b_{t-1}^0.$ |

| |
|--|
| $a^1 \leftarrow b_{t-2}^1, b_{t-1}^1.$ $b^1 \leftarrow a_{t-2}^0, b_{t-2}^0.$ $a^0 \leftarrow b_{t-1}^0.$ $a^0 \leftarrow b_{t-2}^1.$ $b^0 \leftarrow a_{t-1}^1.$ $b^0 \leftarrow a_{t-2}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow a_{t-2}^0.$ |
|--|

Output after processing all possible traces

| |
|--|
| $a^1 \leftarrow b_{t-2}^1, b_{t-1}^1.$ $b^1 \leftarrow a_{t-2}^0, b_{t-2}^0.$ $a^0 \leftarrow b_{t-1}^0.$ $a^0 \leftarrow b_{t-2}^1.$ $b^0 \leftarrow a_{t-1}^1.$ $b^0 \leftarrow a_{t-2}^0.$ $b^1 \leftarrow a_{t-1}^1.$ $b^1 \leftarrow a_{t-2}^0.$ |
|--|

TABLE 4.2: Execution of **LFkT** on some traces of the figure 4.4

On this example, the algorithm learns only 1-step relations at the beginning. It initializes the program P'_1 with the most general hypotheses, i.e. fact rules. The algorithm then analyzes the first trace $10 \rightarrow 01 \rightarrow 01$. From this trace, it extracts a 2-step interpretation transition: $(a_{t-2}^1 b_{t-2}^0 a_{t-1}^0 b_{t-1}^1, a^0 b^1)$. And a 1-step interpretation transition: $(a_{t-1}^0 b_{t-1}^1, a^0 b^1)$ that represents the end of the trace. From $(a_{t-2}^1 b_{t-2}^0 a_{t-1}^0 b_{t-1}^1, a^0 b^1)$, since a^0 and b^1 are present in the second interpretation, the algorithm infers anti-rules for a^1 and b^0 : $R_{a^1} := a^1 \leftarrow a_{t-1}^0, b_{t-1}^1$ and $R_{b^0} := b^0 \leftarrow a_{t-1}^0, b_{t-1}^1$. The atoms referring to time step $t-2$ are ignored by the algorithm in the revision of P'_1 , since only 1-step relations are considered in this program. R_{a^1} and R_{b^0} are used to revise the rules of P'_1 that subsume them, so that these rules become consistent with the interpretation. This is done using least specialization, by the addition of literals into the body of the rules. Those added literal are represented in bold in the table. Here, the fact rule of a^1 and b^1 are revised.

From $(a_{t-1}^0 b_{t-1}^1, a^0 b^1)$ it infers an anti-rule of a^1 that is $R'_{a^1} := a^1 \leftarrow a_{t-1}^0, b_{t-1}^1$ and an anti-rule of b^0 that is $R'_{b^0} := b^0 \leftarrow a_{t-1}^0, b_{t-1}^1$. The rules of P'_1 are revised by R'_{b^0} and R'_{a^1} . Here, the rule previously revised are revised again. The program then becomes consistent with the trace $11 \rightarrow 01 \rightarrow 01$.

Now, **LFkT** analyzes the trace $11 \rightarrow 01 \rightarrow 10$. This trace is not 1-consistent with the previous trace $11 \rightarrow 01 \rightarrow 10$, e.g. from the state 01 there are two possible next states. A new program P'_2 is initialized with minimal rules that consider 2-step relations. P'_2 is then revised to be consistent with the previous traces stored in O' . The program is revised by the 2-step interpretation of $11 \rightarrow 01 \rightarrow 10$. Here, some rules are subsumed after specialization and are then removed. In the table they are stroked, like $a^1 \leftarrow a_{t-2}^1, b_{t-2}^1$ that is subsumed by $a^1 \leftarrow b_{t-2}^1$.

Solving continues until all traces have been analyzed. At the end, the programs learned are merged into a unique one while ensuring that there is no subsumption between rules. Here for the sake of space and readability, we exhibit only the processing of 4 traces. To ensure that the output is correct, it will need to analyze all possible traces of the system that is d^{n*k} , with n being the number of variables of the system, d their domain size and k the delay of the system. Here, it will require the analysis of all 16 possible traces. If all possible traces are processed by the algorithm, it will output a logic program that is exactly the Markov(k) system which produced those traces. Here the rules outputted will be the ones of the system of Example 4.6. In the table, they are highlighted in blue the first time they appear. In this case study, the processing of remaining traces will lead to the specialization and deletion of the other rules.

4.3.4 Evaluation

In this section, we evaluate our new learning algorithm through experiments. We apply **LFkT** to learn Boolean networks from biological literature. These Boolean networks are taken from

Dubrova and Teslenko [77]: these networks model the control of flower morphogenesis in *Arabidopsis thaliana*, the budding yeast cell cycle regulation, the fission yeast cell cycle regulation and the mammalian cell cycle regulation. Those benchmark were originally Boolean Markov(1) systems. Here, *LFkT* will learn rules for both value of each variable (0 and 1). In these experiments, we artificially introduced delays in the relation between the variables to obtain Markov(k) system. This is done by arbitrary modifying some rules conditions from $t - 1$ by $t - k$.

| Runtime (in seconds)/Output size (# of rules) | | | | |
|---|-------------------|------------------|---------------|------------------|
| Delay | Mammalian (10) | Fission (10) | Budding (12) | Arabidopsis (15) |
| k=1 | 0.06s / 23 | 0.07s / 24 | 0.42s / 54 | 5.87s / 28 |
| k=2 | 6.92s / 3,922 | 2.5s / 2,003 | 213s / 10,551 | 6,902s / 4,981 |
| k=3 | 752s / 54,692 | 49.39s / 11,312 | 20,970s | T.O. |
| k=4 | 12,448s / 293,020 | 302s / 32,581 | T.O. | T.O. |
| k=5 | T.O. | 1,197s / 70,322 | T.O. | T.O. |
| k=6 | T.O. | 3,585s / 129,603 | T.O. | T.O. |
| k=7 | T.O. | 9,401s / 216,212 | T.O. | T.O. |

FIGURE 4.5: Runtime and output size of **LFkT** on learning the benchmark from [77], varying the delay of the system.

Figure 4.5 shows the evolution of runtime of learning with **LFkT** on four Boolean networks of [77]. The number of variables in the benchmark goes from 10 to 15, time limit is set to 10 hours. The four Boolean networks were implemented as a logic program using Answer Set Programming [92]. Traces of executions of these programs have been computed using the Answer Set Solver `clasp` [86]. For each possible initial state, the ASP program generates the corresponding trace according to each variable rules. In each experiment, the input of **LFkT** is 2^n traces of size k with n the number of variables of the benchmark and k its delay.

All experiments are run with a C++ implementation of **LFkT** on a processor Intel Xeon (X5650, 2.67GHz) with 12GB of RAM. The main purpose of these experiments is to analyze the impact of the delay associated to the model on the global runtime. The ASP source code of each benchmark and the corresponding output of **LFkT** for these experiments can be accessed as text files at

<http://tony.research.free.fr/paper/CMSB/experiment.zip>.

In the table of Figure 4.5, the evolution of runtime shows that, in practice, considering a higher delay for a given system increases exponentially the learning time of **LFkT** for this system. But the importance of this combinatorial explosion highly depends on the rules of the system that is learned. For the mammalian cell benchmark, runtime is multiplied by about 100 when increasing the delay from 1 to 2 or 2 to 3. Similar results are observed in the case of a much bigger network that is the arabidopsis thaliana benchmark. But in the case of the budding yeast benchmark, increasing the delay leads to a even higher runtime explosion: from delay of 1 to 2, learning time increases by 500, but from 2 to 3 runtime increases by about 100. On the other hand, for the fission yeast benchmark, runtime is only multiplied by respectively 35 and 20 when increasing the delay from 1 to 2 and 2 to 3. Again, for the mammalian cell benchmark increasing the delay from 3 to 4 only increase runtime by 16. Those results show that in practice, the time

required for learning a system highly depends on the dynamics of the system. Learning the dynamics of the mammalian cell takes more time than learning the ones of the fission yeast, even though the system has the same number of variables (10). And the influence of the evolution of the delay of the system does not have the same impact on runtime for these two benchmarks.

In those experiments, the output of **LFkT** is an over-approximation of the original dynamics of the system, in the sense that all original rules of the learned system are either present in the output or subsumed by some rules of the output. For example, learning the Markov(2) version of the fission yeast benchmark leads to 2,003 rules. Here, all 24 original rules of the benchmark are present in the output but more traces are necessary to process in order to eliminate the other rules. Even if over-approximating the dynamics may lead to be over-pessimistic when checking some properties, it is still useful for two main reasons: (i) If a dynamics property is proven false on the over-approximation, then it is false on the real system; (ii) The output of **LFkT** may help to design new practical experiments in order to validate or refute some rules.

4.3.5 Conclusion

In this section, 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 initially 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 [91]. 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.

4.4 Asynchronous Systems

In this section we provide an extension of our framework to model and learn asynchronous systems. In [100], A. Garg et al. address the differences and complementarity of synchronous and asynchronous semantics to model regulatory networks and identify attractors. The authors focus on attractors, which are central to gene regulation. Indeed they give some precious information about the cell differentiation processes. But previous studies about attractors with synchronous semantics [101, 102] and asynchronous semantics [103, 104] showed that different updating rules result in different attractors. The benefits of the synchronous model are to be computationally tractable, while classical state space exploration algorithms fail on asynchronous ones. Yet the synchronous modeling relies on one quite heavy assumptions: all genes can make a transition simultaneously and need an equivalent amount of time to change their expression level. Even if this is not realistic from a biological point of view, it is usually sufficient as the exact kinetics and order of transformations are generally unknown. The asynchronous semantics however helps to capture more realistic behaviors. At a given time, a single gene can change its expression level. This results in a potential combinatorial explosion of the number of states. To illustrate this issue, the authors of [100] compare the time needed to compute the attractors of various models (mammalian cell, T-helper, dendritic cell, . . .) and compare the results with synchronous and asynchronous semantics. To make the asynchronous computation be more efficient, they propose a new algorithm to capture the attractors, that is based on a combination of the synchronous and asynchronous ones. They identify the cases when attractors of the asynchronous model match with the associated synchronous model. This has been implemented in a software package called genYsis, which performs better than other existing approaches, especially because it takes benefit from the aforementioned combined approach.

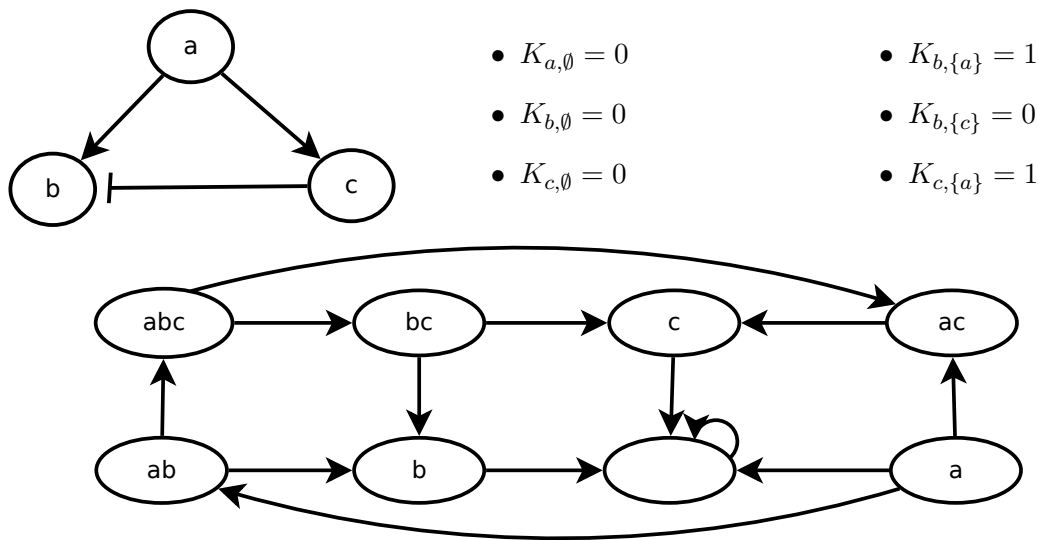


FIGURE 4.6: The feed-forward loop of [1]: influence graph (top-left), update rules (top-right) and the corresponding transition diagram (bottom).

Example 4.8. Figure 4.6, shows an example of asynchronous Boolean network. Here the three firsts transition rule of the form $K_{X,\emptyset} = 0$ means that the variable X can be inhibited at the next state if there is no influence on it. The rule $K_{b,\{a\}} = 1$ means that b can be activated at the next state if a is present in the current state. The rule $K_{b,\{c\}} = 0$ means that b can be inhibited at the next state if c is present in the current state.

4.4.1 Time delays in asynchronous framework

In [105], Thomas and Kaufman discuss the meaning of time delays in asynchronous Thomas' networks. Time delays constitute a bridge with the ordinary differential equations formalism in the sense that they relate to the kinetic parameters involved in the production and decay of a biological component. Compared to the logical description historically used, time delays bring additional information. Without time delays, it was possible to exhibit cycle and the corresponding periodicity, but not to determine whether this was a stable or unstable cycle. In their paper, the authors propose a brute-force method to induce graphs from a partial state table. Thanks to a Boolean algebra-based analysis, it is possible to propose some models that can represent some given biological properties of the model to infer (e.g., stable states or cycles).

4.4.2 Formalization

An asynchronous Boolean network can be represented by a set of rules of the following forms:

$$activate(p) \leftarrow \neg p \wedge p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n$$

$$inhibit(p) \leftarrow p \wedge p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n$$

$$no_change \leftarrow p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n$$

where p and p_i 's are atoms ($n \geq m \geq 1$) and $activate, inhibit, no_change$ are predicates.

Definition 4.20 (Asynchronous successors). Let I be the interpretation of the current state of an asynchronous Boolean network B represented by a set of rules S . Let $TP(I, S) = \{h(R) | R \in S, b(R)^+ \subseteq I, b(R)^- \cap I = \emptyset\}$. The successors of I in B according to S is

$$next(I, S) = \{I \cup \{p\} | activate(p) \in TP(I, S)\} \cup \{I \setminus \{p\} | inhibit(p) \in TP(I, S)\} \cup \{I | no_change \in TP(I, S)\}$$

Example 4.9. The asynchronous Boolean network of Figure 4.6, can be represented as follow.

- $K_{a,\emptyset} = 0$ by $inhibit(a) \leftarrow a$.
- $K_{b,\{a\}} = 1$ by $activate(b) \leftarrow a \wedge \neg b$.
- $K_{b,\emptyset} = 0$ by $inhibit(b) \leftarrow \neg a \wedge b \wedge \neg c$.
- $K_{b,\{c\}} = 0$ by $inhibit(b) \leftarrow b \wedge c$.
- $K_{c,\emptyset} = 0$ by $inhibit(c) \leftarrow \neg a \wedge c$.
- $K_{c,\{a\}} = 1$ by $activate(c) \leftarrow a \wedge \neg c$.

A multi-valued asynchronous system can be represented by a set of multi-valued rules where each rules have the form:

$$change(v^{val}) \leftarrow v^{val'} \wedge p_1 \wedge \dots \wedge p_n.$$

$$no_change \leftarrow p_1 \wedge \dots \wedge p_n.$$

where $v^{val}, v^{val'}, p$ and p_i 's are anoted atoms ($n \geq 1, val \neq val'$) and $change$ is a predicate.

Definition 4.21 (Multi-valued Asynchronous successors). Let I be the current state of an asynchronous system represented by a set of multi-valued rules S . Let $TP(I, S) = \{h(R) | R \in S, b(R) \subseteq I\}$. The successors of I in B according to S is

$$next(I, S) = \{I \setminus \{v^{val'}\} \cup \{v^{val}\} | change(v^{val}) \in TP(I, S), v^{val'} \in I\} \cup \{I | no_change \in TP(I, S)\}$$

Example 4.10. The asynchronous system of Figure 4.6, can be represented as follow.

- $K_{a,\emptyset} = 0$ by $change(a^0) \leftarrow a^1$.
- $K_{b,\{a\}} = 1$ by $change(b) \leftarrow a^1 \wedge b^0$.
- $K_{b,\emptyset} = 0$ by $change(b^0) \leftarrow a^0 \wedge b^1 \wedge c^0$.
- $K_{b,\{c\}} = 0$ by $change(b) \leftarrow b^1 \wedge c^1$.
- $K_{c,\emptyset} = 0$ by $change(c^0) \leftarrow a^0 \wedge c^1$.
- $K_{c,\{a\}} = 1$ by $change(c) \leftarrow a^1 \wedge c^0$.

4.4.3 Algorithms

We now adapt the **LFIT** framework to learn asynchronous systems. As a first step we do not consider delay and show how to simply adapt *LF1T* when variables are Boolean. In this case, the only real difference with learning synchronous system, is the rule that are infered from the transitions.

In the transition of an asynchronous system, there is only one difference between the current state and the next state. Here the idea is to only learn this change. When variable are Boolean, this change can be either an activation or a inhibition: a variable change its value from false to true or from true to false. Here, we need to learn rules for both changes.

Like in previous versions, **LFIT** takes a set of state transitions E as input and outputs an NLP P that realizes E .

4.4.3.1 Learning using generalization

To learn Boolean network using generalization, the algorithm starts with an empty set of rules and will iteratively analyzes each transition $(I, J) \in E$:

- If $\exists A \in J, A \notin I$ the transition (I, J) is an activation of A and **LF1T** infers a rule R_A^I :

$$R_A^I := \text{activate}(A) \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$$

- If $A \in I, A \notin J$ the transition (I, J) is an inhibition of A and **LF1T** infers a rule R_A^I :

$$R_A^I := \text{inhibit}(A) \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$$

- If $I = J$, the transition is a self transition (I, I) and **LF1T** infers a rule R :

$$R := \text{no_change} \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$$

Algorithm 22 show the pseudo-code of $\{\text{LF1T}\}$ for learning asynchronous Boolean systems. The way that rules are added into the program learned is the same as before (line 15, `AddRule` is exactly Algorithm 26).

Algorithm 22 **LF1T**(E, P) with generalization for asynchronous Boolean systems

```

1: INPUT: a set  $E$  of pairs of Herbrand interpretations and an NLP  $P$ 
2: OUTPUT: an NLP  $P$ 

3: while  $E \neq \emptyset$  do
4:   Pick  $(I, J) \in E$ ;  $E := E \setminus \{(I, J)\}$ 
5:   if  $I \neq J$  then
6:     if there is  $A \notin I, A \in J$  then
7:        $R := \text{activate}(A) \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$ 
8:     end if
9:     if there is  $A \in I, A \notin J$  then
10:       $R := \text{inhibit}(A) \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$ 
11:    end if
12:   else
13:      $R := \text{no\_change} \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$ 
14:   end if
15:   AddRule( $R_{v^{val'}}$ ,  $P$ )
16: end while
17: return  $P$ 

```

To learn multi-valued systems using generalization, the algorithm starts with an empty set of rules and will iteratively analyzes each transition $(I, J) \in E$. If there is a variable v that changed its value between I and J , we infer a rule that represent this change:

- If $v^{val} \in I, v^{val'} \in J, val \neq val'$ **LF1T** infers a rule $R_{change(v^{val'})}^I$:

$$R_{v^{val'}}^I := change(v^{val'}) \leftarrow \bigwedge_{B_i \in I} B_i$$

- otherwise **LF1T** infers a rule R :

$$R := no_change \leftarrow \bigwedge_{B_i \in I} B_i$$

The pseudo code of *LF1T* for multi-valued asynchronous system is given in Algorithm 23. The way that rules are added into the program learned is the same as before (line 11, *AddRule* is exactly Algorithm 26).

Algorithm 23 **LF1T**(E, P) with generalization for asynchronous multi-valued systems

```

1: INPUT: a set  $E$  of pairs of Herbrand interpretations and an NLP  $P$ 
2: OUTPUT: an NLP  $P$ 

3: while  $E \neq \emptyset$  do
4:   Pick  $(I, J) \in E$ ;  $E := E \setminus \{(I, J)\}$ 
5:   if  $I \neq J$  then
6:     Let  $v^{val} \in I, v^{val'} \in J, val \neq val'$ 
7:      $R := change(v^{val'}) \leftarrow \bigwedge_{B_i \in I} B_i$ 
8:   else
9:      $R := no\_change \leftarrow \bigwedge_{B_i \in I} B_i$ 
10:  end if
11:  AddRule( $R, P$ )
12: end while
13: return  $P$ 

```

4.4.3.2 Learning using specialization

To learn using specialization, we also need to learn the change but the way its done is different. The rules inferred from the transitions are anti-rule, that can be consider as counter example. Activation is a counter example of inhibition and inhibition is a counter example of Activation. By definition, its not possible to have both activation and inhibition of the same variable from a given state. Intuitively, we could think that we can iteratively learn activation (resp. inhibition) by specialization: each time we observe an inhibition (resp. activation) we specialize our knowledge. But, when a variable does not change its value, its also a counter example for both activation and inhibition. And its possible that from the same state, we have in case 1 an activation of a variable a and in case 2 the change of another variable b . If we specialize the rule of activation of a by the case 2, we do not subsume the case 1 any more. So, learning with specialization cannot be done in this way.

Specialization works for synchronous system, because what we try to learn is always true, a rule is always applied if it match the current state. But, in an asynchronous system, we try to learn possibilities. We want to know when a variable **can** take the value true or false. A solution to this problem is to learn the inverse of what we want. The idea is to learn, when the variable **cannot** take the value true (resp. false). In this case, when we observe that the variable take the value true, its a real counter example of when the variable take the value false. Here, the extension to multi-valued variable is straight forward: For each variable values, we learn when the variable **cannot** take this value. Each time we observe that a variable take the value v , we specialize the rules that say that the variable cannot take the value v . At the end, we only need to compute the inverse logic program.

Algorithm 24 LF1T(E) with specialization for asynchronous multi-valued systems

```

1: INPUT: a set  $E$  of pairs of multivalued interpretations and an NLP  $P$ 
2: OUTPUT: An NLP  $P$  such that  $J = T_P(I)$  holds for any  $(I, J) \in E$ .

3:  $P$  a NLP
4:  $P := \emptyset$ 

5: // 1) Initialize  $P'$  with the most general logic program
6: for each atom  $v^{val} \in \mathcal{B}$  do
7:   for each atom  $v^{val'} \in \mathcal{B}, val \neq val'$  do
8:      $P := P \cup \{v^{val} \leftarrow v^{val'}\}$ 
9:   end for
10: end for
    // Specify  $P$  by interpretation of transitions
11: while  $E \neq \emptyset$  do
12:   Pick  $(I, J) \in E$ ;  $E := E \setminus \{(I, J)\}$ 
13:   if  $\exists v^{val} \in J, v^{val} \notin I$  then
14:      $R_{v^{val}}^I := change(v^{val}) \leftarrow \bigwedge_{B_i \in I} B_i$ 
15:   else // Self-transition  $(I, I)$ 
16:      $R := no\_change \leftarrow \bigwedge_{B_i \in I} B_i$ 
17:   end if
18:    $P := \text{Specialize}(P, R)$ 
19: end while
20: return  $\neg P$ 

```

For this algorithm we consider a multi-valued logic program representation. Like before, the algorithm starts with fact rules and will iteratively analyzes each transition $(I, J) \in E$. If there is a variable v that changed its value between I and J , we infer an anti-rule that represent this change: **LF1T** infers an anti-rule $R_{v^{val}}^I$:

$$R_{v^{val'}}^I := change(v^{val'}) \leftarrow \bigwedge_{B_i \in I} B_i$$

If $I = J$, **LF1T** also infers an anti-rule of *no_change*, R :

$$R := no_change \leftarrow \bigwedge_{B_i \in I} B_i$$

The pseudo code of $LF1T$ for asynchronous multi-valued system is given in Algorithm 24. The way that rules are added into the program learned is the same as before (line 18, *Specialize* is exactly Algorithm 21).

The same method can be used to capture delayed influence of asynchronous system. Here, the delay of the system needs to be known in advance (like in Section 4.1.2). Since asynchronism leads to inconsistent traces, we cannot determine if an inconsistency is the consequence of a delays or asynchronism.

4.5 Uncertainty

In this section we extend the *LFIT* framework to learn probabilistic dynamics by proposing an extension of *LFIT* for learning from uncertain state transitions. Where other works like [106] perform inferences from a probabilistic logic program, what we do is inferring the rules of such logic program. The programs inferred by our new algorithm are similar to paraconsistent logic program of [99]. The use of annotated atoms allows the learned programs to induce multiple values for the same represented variable. It allows us to represent multi-valued models and capture non-deterministic state transitions. Our semantics differs from previous work like [107]. Here, the authors consider probabilistic logic programs as logic programs in which some of the facts are annotated with probabilities. But in our method, it is the rules that have probabilities and they are independent.

4.5.1 Formalization

An non-deterministic system can be represented by a set of logic programs where the rules have the following form:

$$R = var^{val} \leftarrow var_1^{val_1} \wedge \dots \wedge var_n^{val_n}(i, j)$$

where var^{val} and $var_i^{val_i}$ are atoms ($n \geq 1$), var^{val} is the head of R again denoted $h(R)$ and i, j are natural numbers, $i \leq j$. Let I be a multi-valued interpretation. R means that i times over j , var takes the value val in the successor of I if $b(R) \subset I$.

Definition 4.22 (Non-deterministic successors). Let I be the multi-valued interpretation of a state of a non-deterministic system S represented by a set of logic programs P . Let P' be a logic program, one of the successors of I according to P' is

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

The set of *successors* of I in S according to P is

$$successor(I, P) = \{J | J \in next(I, P_i), P_i \in P\}$$

Definition 4.23 (Successor Probability). Let I and J be the multi-valued interpretation of two states of a probabilistic system PS represented by a set of set of rules S . The probability for J to be the successor of I is 0 if $J \notin successor(I, S)$. Otherwise, let $S_i \in S$, $next(I, S_i) = \{h(R) | R \in S, b(R) \subseteq I\}$. If $\exists S_i \in S$ such that $J = next(I, S_i)$, the probability for J to be the successor of I is

$$\prod_{var^{val} \in J} max(i/j), (R = var^{val} \leftarrow b(R)(i, j)) \in P(I, S_i)$$

4.5.2 Algorithm

We now present **LUST**, an extension of *LFIT* for Learning from Uncertain State Transition. **LUST** learns a set of deterministic logic programs. The main idea is that when two transitions are not consistent we need two different programs to realize them. The first program will realize the first transition and the second one will realize the second transition. The algorithm will output a set of logic programs such that every transition given as input is realized by at least one of those programs. The rules learned also provide the probability of the variable values in the next state. The probability of each rule $R = var^{val} \leftarrow b(R)(i, j)$ is simply obtained by counting how many transitions (I, J) it realizes (when $b(R) \subseteq I$ and $var^{val} \in J$), represented by i , over how many transitions it matches (when $b(R) \subseteq I$), represented by j .

LUST

- **Input:** a set of pairs of interpretations E and a set of atoms \mathcal{B} .
- Step 1: Initialize a set of logic programs P with one program P_1 with fact rules for each atom of \mathcal{B} .
- Step 2: Pick (I, J) in E , check consistency of (I, J) with all programs of P :
- if there is no logic program in P that realizes (I, J) then
 - copy one of the logic programs P_i into a P'_i and add rules in P'_i to realize (I, J) .
 - Use full ground resolution to generalize P'_i .
- Step 3: Revise all logic programs that realize (I, J) by using least specialization.
- Step 4: If there is a remaining transition in E , go to step 2.
- Step 5: Compute the probability of each rule of all programs P_i according to E .
- **Output:** P a set of multi-valued logic programs that realizes E .

The detailed pseudo-code of the *LUST* algorithm is given in Algorithm 25. The algorithm starts with one logic program that contains all fact rules. Each input transition is analyzed one by one. If no program can realize the observed trace, one is copied and rules are added into this copy so that it realize the transition. The program that realizes T are then revised using least specialization like in previous version of the algorithm. The program that does not realize the transition realizes another one previously observed that is not consistent with the new one because of the non-determinism of the system. Those program cannot be specialized by this transition because we will lost the information of the previous transition it realizes. Finally, the algorithm will output a set of logic program, each one of them realizes some traces of O and all traces of O are realized by at least one of the program.

Algorithm 25 LUST(E, \mathcal{B})

```

1: INPUT: a set of pair of interpretations  $E$  and a set of atoms  $\mathcal{B}$ 
2: OUTPUT:  $P$  a set of logic programs

3:  $E' := \emptyset$ 
4:  $P := \emptyset$ 
   // Initialize  $P$  with one program with the most general rules
5:  $P_1 := \emptyset$ 
6: for each  $var^{val} \in \mathcal{B}$  do
7:    $P_i := P_i \cup \{var^{val} \leftarrow\}$ 
8: end for
9:  $P := P \cup P_1$ 
   // 2) Revise  $P$  to realize every transition
10: while  $E \neq \emptyset$  do
11:   Pick  $(I, J) \in E$ ;  $E := E \setminus \{(I, J)\}$ 
12:   // 2.1) Check if  $(I, J)$  is realizable
13:   for each logic program  $P_i$  of  $P$  do
14:      $realize\_I\_J := true$ 
15:     for each  $var^{val} \in J$  do
16:       if  $\exists R \in P_i, b(R) \subseteq I, h(R) \in J$  then
17:          $realize\_I\_J := false$ 
18:       end if
19:     end for
20:     if  $realize\_I\_J = true$  then
21:       break
22:     end if
23:   end for
   // 2.2) construct a logic program that realize  $(I, J)$ 
24:   if  $realize\_I\_J = false$  then
25:     for each  $var^{val} \in J$  do
26:        $R := var^{val} \leftarrow \bigwedge_{B_i \in I} B_i(0, 0)$ 
27:       choose a  $P_i \in P$ 
28:        $P'_i := P_i$ 
29:        $P := \text{AddRule}(P'_i, R, \mathcal{B})$ 
30:     end for
31:   end if
   // 3) revise logic programs that realize  $(I, J)$ 
32:   for each logic program  $P_i$  of  $P$  do
33:     for each  $var^{val} \in J$  do
34:       for each  $var^{val'} \in \mathcal{B}, val' \neq val$  do
35:          $R^I_{var^{val'}} := var^{val} \leftarrow \bigwedge_{l_i \in I} l_i(0, 0)$ 
36:          $P_i := \text{Specialize}(P_i, R^I_{var^{val'}}, \mathcal{B})$ 
37:       end for
38:     end for
39:   end for

40:    $E' := E' \cup (I, J)$  // 4) Remember  $(I, J)$  and continue with other transitions
41: end while
   // 5) Compute the likelihood of each rules
42: for each logic program  $P_i$  of  $P$  do
43:   for each rule  $R \in P_i$  such that  $h(R) = var^{val} \leftarrow b(R)(i, j)$  do
44:     for each  $(I, j) \in E'$  do
45:       if  $b(R) \subseteq I$  then
46:          $R := var^{val} \leftarrow b(R)(i, j + 1)$ 
47:         if  $h(R) \in J$  then
48:            $R := var^{val} \leftarrow b(R)(i + 1, j)$ 
49:         end if
50:       end if
51:     end for
52:   end for
53: end for
54: return  $P$ 

```

Algorithm 26 AddRule(R, P) (with multi-valued ground resolution)

```

1: INPUT: a rule  $R$  and a NLP  $P$ 

2: // 1) Check subsumptions
3: for each rule  $R_P$  of  $P$  do
4:   if  $R$  is subsumed by  $R_P$  then
5:     return  $P$ 
6:   end if
7:   if  $R$  subsumes  $R_P$  then
8:      $P := P \setminus \{R_P\}$ 
9:   end if
10: end for
11: // 2) Produce generalizations
12: for each  $var^{val} \in R$  do
13:   if  $R$  can be generalized by  $P$  on  $var^{val}$  then
14:      $generalized := true$ 
15:      $P := AddRule(R', P)$ 
16:   end if
17: end for
18: if  $generalized = true$  then
19:   return  $P$ 
20: else
21:   return  $P \cup R$ 
22: end if

```

Algorithm 27 specialize(P, R, \mathcal{B}) : specialize P to avoid the subsumption of R

```

1: INPUT: a logic program  $P$  and a rule  $R$ 
2: OUTPUT: the least specialization of  $P$  by  $R$ .

3:  $conflicts$  : a set of rules
4:  $conflicts := \emptyset$ 
   // Search rules that need to be specialized
5: for each rule  $R_P \in P$  do
6:   if  $R_P$  subsumes  $R$  then
7:      $conflicts := conflicts \cup R_P$ 
8:      $P := P \setminus R_P$ 
9:   end if
10: end for
   // Revise the rules by least specialization
11: for each rule  $R_c \in conflicts$  do
12:   for each literal  $var^{val} \in b(R)$  do
13:     if  $var^{val} \notin b(R_c)$  then
14:       for each  $var^{val'} \in \mathcal{B}, val' \neq val$  do
15:          $R'_c := (h(R_c) \leftarrow (b(R_c) \cup var^{val'}))$ 
16:         if  $P$  does not subsume  $R'_c$  then
17:            $P := P \setminus$  all rules subsumed by  $R'_c$ 
18:            $P := P \cup R'_c$ 
19:         end if
20:       end for
21:     end if
22:   end for
23: end for
24: return  $P$ 

```

4.5.3 Learning Probabilistic Action Models

In this section we present the application and the evaluation of the *LUST* algorithm to learn action model in robotic applications. Here we present a conjoint work with David Martinez who was internship student at Inoue lab from February to August 2015. Our approach learns an action model encoded as a set of planning operators. Each operator describes how the value of a predicate changes based on a set of preconditions given that an action is executed. We present a novel method to learn in two levels as shown in Fig. 4.7.

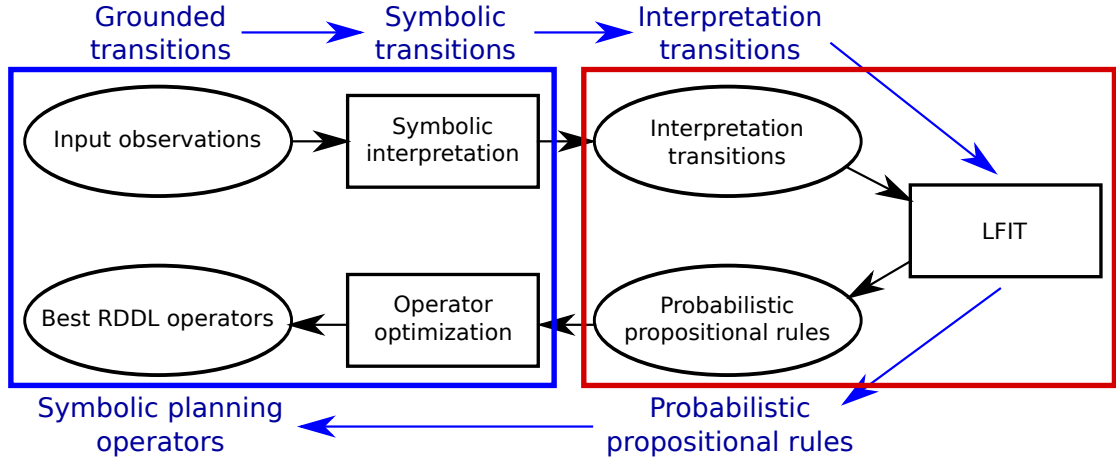


FIGURE 4.7: Overview of the learning framework. The modules that interact with planning data and operators are in the blue rectangle on the left, while the modules related to logic programming are included inside the red rectangle on the right. The input of the method are grounded transitions, which are then converted to symbolic transitions to generalize between different objects. To obtain rule candidates, *LFIT* requires that the symbolic transitions are represented with propositional atoms. Finally, after rule candidates are obtained, they are transformed to planning operators to select the best set of operators.

On the right, our framework is used to generate rules. Given a set of state transitions of a system, it can learn a normal logic program that captures the system dynamics. The resulting rules are all candidates that have to be considered to select the best planning operators. On the left, the data is generalized between different objects by using a relational representation, and an optimization method is used to select the best subsets of planning operators that explain input transitions while maintaining generality. The dependency relations between the planning operators are used to efficiently select the best candidates. Our method is designed to learn RDDL-like [108] operators, where each variable is updated separately based on a set of preconditions. To that end, in this section we present the following novel improvements:

- The integration of logic programming to efficiently limit the number of rule candidates with a relational representation that provides better generalization, and an optimization method to select the best subsets of planning operators.
- An optimization method that makes use of the dependency relation between rules to select subsets of planning operators efficiently. It uses a score function that balances the

likelihood of the operators and their generality. Moreover, the confidence is also used so that generality is strongly preferred for uncertain estimations.

4.5.3.1 Integration of Logic Programming and Planning Domains

In this section we describe the formalization used to represent planning operators, as well as the data conversions needed to learn symbolic operators from grounded transitions while using propositional logic programming.

4.5.3.2 Planning Model

Although *LUST* uses a propositional representation, the planning model uses a relational representation to provide a better generalization between different states. Relational domains represent the state structure and objects explicitly. These domains are described by using a vocabulary of predicates \mathcal{P} and actions \mathcal{A} , and a set of objects C_π . Predicates and actions take objects as arguments to define their grounded counterparts.

Example 4.11. Let $on(X,Y)$ be a symbolic predicate, and $\{box1, box2, box3\}$ be a set of objects. Three possible groundings of the symbolic predicate $on(X,Y)$ with the given atoms are $on(box1, box2)$, $on(box1, box3)$ and $on(box3, box1)$.

A grounded predicate or action is equivalent to an atom. In a planning domain, a planning state s is a set of grounded predicates $s = \{p_1, p_2, \dots, p_n\}$ that is equivalent to an Herbrand interpretation.

Our planner operators represent a subset of RDDDL domains [109]. For each variable, we define the probability that it will become true in the next state based on a set of preconditions. In contrast to the full RDDDL specification, these preconditions can only consist of an action and a set of predicates. Thus we define a planning operator as a tuple $o = \langle o_{p^{val}}, o_{act}, o_{prec}, o_{prob} \rangle$ where:

- $o_{p^{val}}$ is a predicate p whose value can change to val by applying the operator. It is equivalent to the head of a logic rule.
- o_{act} is the action that has to be executed for the operator to be applicable.
- o_{prec} is a set of predicates that have to be satisfied in the current state so that the planning operator is applicable. It is equivalent to the body of a logic rule.
- o_{prob} is the probability that p^{val} will be true.

Note that planning operators use a relational representation (i.e. symbolic predicates), while rules learned by *LUST* are propositional (i.e. annotated atoms, which are equivalent to grounded predicates). Moreover, a symbolic planning operator can be grounded by replacing each variable by an object.

4.5.3.3 Data Representation

To have a more general and compact model, we are using a relational representation at the planning level. The input of our method consists of state transitions that are tuples $t = \langle s, act, s' \rangle$ where s and s' are the states before and after executing the action act . The states consist of sets of grounded predicates, and act is a grounded action. On the other hand, the output is a set of symbolic (i.e. non-grounded) planning operators. Therefore, our method transforms initial grounded data to symbolic planning operators. Moreover, *LUST* works with propositional atoms, so a transformation from symbolic predicates to atoms and back to symbolic predicates is also needed. Figure 4.7 shows the needed data conversions, which are explained in more detail below.

Transform grounded transitions to symbolic transitions:

- **Input:** a set of grounded transitions $T = [t_1, t_2, \dots, t_n]$.
- For each transition $t = \langle s, act, s' \rangle$:
 - Take every argument χ of the action act .
 - Substitute χ for a default symbolic parameter in s , act , and s' .
 - Create a new transition t' with the symbolic predicates in s and s' and the symbolic action act .
 - Add the new transition t' to T' .
- **Output:** set of symbolic transitions T' .

Transform a symbolic transition to an interpretation transition :

- **Input:** a symbolic transition $t = \langle s, act, s' \rangle$.
- Assign an atom to each predicate in s , act and s' .
- I = atoms that correspond to s .
- Add act as an atom in I that represents the action.

- J = atoms that correspond to s' .
- **Output:** interpretation transition (I, J) .

To transform a planning symbolic transition to an interpretation transition, a labeled atom that encodes the action is added to the body of the interpretation transition. As each transition has exactly one action, a multi-valued variable represents it more efficiently than boolean, otherwise every action would have to be represented as different variables with only one being true. Moreover, each symbolic predicate value is represented by one labeled atom. After the logic programs are generated, the labeled atoms are translated back to symbolic predicates by using the same conversion.

Transform a logic program to a set of planning operators:

- **Input:** a logic program P .
- For every rule $R \in P$ such that $h(R) = var^{val} \leftarrow b(R)(i, j)$, a planning operator o is created so that:
 - $o_{pval} = var^{val}$.
 - $o_{act} =$ the action in the atoms of $b(R)$.
 - $o_{prec} =$ the set of atoms in $b(R)$ that represent predicates.
 - $o_{prob} = i/j$.
 - Add o to \mathcal{O}
- **Output:** set of planning operators \mathcal{O} .

4.5.3.4 Selecting the Set of Planning Operators

In this section we present the method to select the best subset of probabilistic planning operators by using the set of logic programs generated by *LFIT*. First, the requirements that the planning operators have to satisfy are presented. Afterwards, we explain the preferences to decide which are the best planning operators. Finally, the method to obtain the desired planning operators is described.

4.5.3.5 Planning Operators Requirements

Probabilistic planners require that only one planning operator can be applied in each state-action pair. Therefore the planning model has to be defined with a set of non-conflicting planning operators, so the planner can always decide which operator to apply for each state-action pair.

Definition 4.24 (Conflicting planning operators). Let o_1 and o_2 be two planning operators that represent the same action $o_{1,act} = o_{2,act}$ and change the same predicate $o_{1,pval} = o_{2,pval}$ with different probabilities $o_{1,prob} \neq o_{2,prob}$. A conflict exists between both planning operators if $\exists s \mid o_{1,prec} \subset s, o_{2,prec} \subset s$.

4.5.3.6 Score Function

LUST provides the minimal set of rules that describe all possible transitions. However, the best subset of non-conflicting planning operators has to be selected to create the model. To decide which are the best set of operators the algorithm uses a score function. The algorithm prefers operators with a high likelihood (i.e. that can successfully explain the state transitions), but also has a regularization term to avoid overfitting to the training data. The regularization is based on the number of planning operators and preconditions in those operators, so that general operators are preferred when the likelihood of both is similar. This regularization penalty is bigger when there are few training transitions to estimate each operator, as general operators are preferred to poorly estimated specific ones. On the other hand, the regularization penalty decreases as our estimate improves with more transitions because the method can be more confident about the operators. The following functions are used to define the score function.

- The likelihood $P(t|\mathcal{O}) = P(s'|s, act, o) \mid (o \in \mathcal{O}, o_{prec} \in s, o_{act} = act)$ is the probability that the transition t is covered by the set of planning operators \mathcal{O} .
- The penalty term $Pen(\mathcal{O}) = |\mathcal{O}| + \frac{1}{|\mathcal{O}|} \sum_{o \in \mathcal{O}} |o_{prec}|$ is the number of planning operators plus the average number of preconditions that they have.
- The confidence in a planning operator $Conf(\mathcal{O}, T)$ is bounded by using the Hoeffding's inequality. The probability that our estimate $\widehat{o_{prob}}$ is accurate enough $|\widehat{o_{prob}} - o_{prob}| \leq \epsilon$ with a number of samples $|T|$ is bounded by $Conf(\mathcal{O}, T) \leq 1 - 2e^{-2\epsilon^2|T|}$.

Finally, the proposed score function is defined as

$$S(\mathcal{O}, T) = \frac{1}{|T|} \left(\sum_{t \in T} P(t|\mathcal{O}) \right) + \alpha \frac{Pen(\mathcal{O})}{Conf(\mathcal{O}, T)} \quad (4.2)$$

where α is a scaling parameter for the penalty term. Also note that $Conf(\mathcal{O}, T) \simeq 1$ when the number of input transitions is large, so the penalty term will always be present to ensure general operators are preferred.

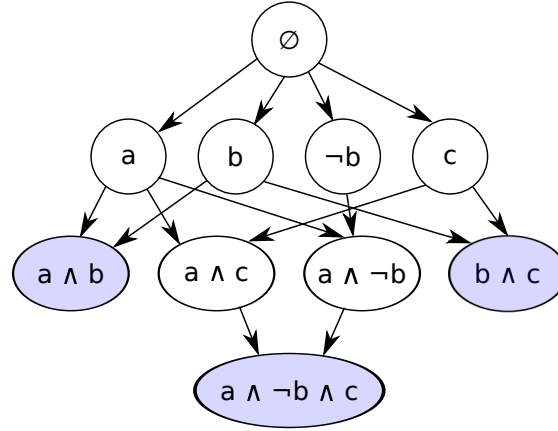


FIGURE 4.8: Example of a parent graph. In each node, the planning operator preconditions are shown. Leaves are the nodes painted in blue.

4.5.3.7 Selecting the Best Planning Operator Subset

In this section we present the algorithm used to select the subset of non-conflicting planning operators that maximizes the score function. To do it efficiently we make use of the dependency relations between operators by using the parent relation graph defined below.

Definition 4.25 (Parent relation). Let o_1 and o_2 be two planning operators that represent the same action $o_{1,act} = o_{2,act}$ and the same effect $o_{1,pval} = o_{2,pval}$. o_1 is a parent of the operator o_2 if $o_{1,prec} \subset o_{2,prec}$.

Definition 4.26 (Parent graph). The parent graph $G_{\mathcal{O}}$ of a set of planning operators $\mathcal{O} = \{o_1, \dots, o_n\}$ is a directed graph with arcs (o_i, o_j) when o_i is parent of o_j and $|o_{j,prec}| - |o_{i,prec}| = 1$. Figure 4.8 shows an example of a parent graph. We will call *leaves* $L(G_{\mathcal{O}})$ of the graph all nodes that do not have a child.

The parent graph orders the rules in levels that represent the generality of the operator: the less predicates in the preconditions the more general the operator is. This graph provides two advantages: generalizing operators is easier as we only have to follow the arcs, and it reduces the number of conflicts to check because general operators won't be checked if other more specific operators that represent the same dynamics exist. An optimal solution can be obtained through backtracking, but as it is computationally intensive, we present the greedy approach that provides similar experimental results and can be applied efficiently in difficult domains with many input transitions.

The first step to select the set of planning operators is to create the parent graph. Afterwards, two algorithms are applied iteratively until no better sets of planning operators can be found. The first algorithm prunes all conflicting leaf operators, leaving only the subset of leaves with the highest score. The second one tries to generalize leaf operators, and removes them from the tree if it can find a better and more general subset.

Prune conflicting leaves:

- **Input:** Parent graph $G_{\mathcal{O}}$ of the set of operator candidates \mathcal{O} that represent one action.
- While leaf nodes $L(G_{\mathcal{O}})$ are conflicting, repeat:
 - Create an undirected graph $G_{conflict}$ that represents the conflicts between $L(G_{\mathcal{O}})$.
 - Calculate the score for each non-conflicting subset in $G_{conflict}$.
 - Remove all leaves from $G_{\mathcal{O}}$ with the exception of the subset with the best score.
- **Output:** Parent graph $G'_{\mathcal{O}}$ with non-conflicting leaves.

Generalize operators:

- **Input:** Parent graph $G'_{\mathcal{O}}$ with non-conflicting leaves.
- For each leaf parent $p \mid \exists(p, l) \in G'_{\mathcal{O}}, l \in L(G'_{\mathcal{O}})$ do:
 - Create a new set of operators with the leaves $\mathcal{O}' = L(G'_{\mathcal{O}})$.
 - Add p to \mathcal{O}' and remove its children leaves $l \mid \exists(p, l) \in G'_{\mathcal{O}}, l \in L(G'_{\mathcal{O}})$.
 - Calculate score $S(\mathcal{O}', T)$.
- If the best parent p improved the score:
 - Remove $l \mid \exists(p, l) \in G'_{\mathcal{O}}, l \in L(G'_{\mathcal{O}})$ from $G'_{\mathcal{O}}$.
- **Output:** Parent graph $G'_{\mathcal{O}}$ with better generalization.

4.5.3.8 Experiments

In this section we provide an experimental evaluation of our approach by learning two domains of the 2014 International Probabilistic Planning Competition (IPPC). The experiments use transitions $t = \langle s, act, s' \rangle$ generated by randomly constructing a state s , randomly picking the arguments of the action act , and then applying the action to generate the state s' . The distribution of samples is biased to guarantee that half of the samples have a chance to change the state. To measure the quality of the learned models, the errors shown represent the differences between the learned operators and the ground truth ones. For each incorrect predicate in an operator preconditions, the number was increased by 1.

The left plot in Fig. 4.9 shows the results obtained in the Triangle Tireworld domain. In this domain, a car has to move to its destination, but it has a probability of getting a flat tire while it moves. The domain has 3 actions. A “Change Tire” action that has only one precondition, a

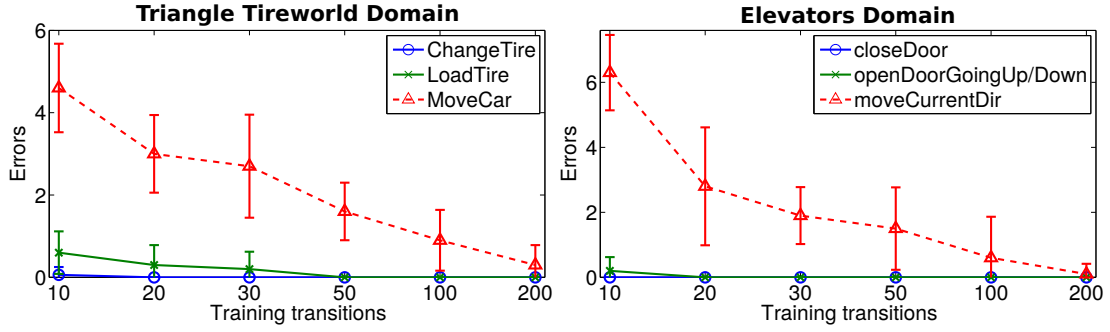


FIGURE 4.9: Results of learning the Triangle Tireworld domain and the Elevators domain from IPPC 2014. The results shown are the means and standard deviations obtained from 10 runs. The number of training transitions per action are shown.

“Load Tire” action that has 2 preconditions to change one predicate, and a “MoveCar” action that changes 3 predicates based on 3 different preconditions. The results show that easy actions can be learned with just a few transitions, while very complex actions require 100 transitions until average errors below 1 are obtained.

The right plot in Fig. 4.9 shows the results obtained in the Elevators domain. In this domain, we are only learning the actions that interact with the elevators, and not the dynamic effects related to people using them. The “openDoorGoing*” and “closeDoor” actions are easy to learn, but the “moveCurrentDir” action has two effects with 3 different preconditions each, and another two effects with 4 preconditions. Therefore, to learn successfully the dynamics of “moveCurrentDir” a large number of input transitions is required.

4.5.4 Conclusions

We have presented a new approach to learn planning operators. In contrast to previous approaches, it learns the dynamics of every predicate independently, and thus can be applied to learn RDDDL domains. The presented approach can find good sets of planning operators efficiently, as it combines logic programming to restrict the set of candidates, and then the parent relations between operators are used to quickly prune bad candidates until the desired subset of operators is found (i.e. a set that generalizes as much as possible while explaining well enough the input transitions).

Chapter 5

Related Works and Comparaison

In this chapter, we present and discuss about related works that share goals and methods with our proposed framework. Section [5.1](#), presents theses works, here we describe what other researcher groups have done recently in our research field. Section [5.2](#), provides a detailed comparison with these works. Here, we show the advantages and weaknesses of our methods regarding these existing works by discussing the possibility and limits of our methods.

5.1 Related Work

5.1.1 Learning from Interpretations

Learning from interpretations [110–112] has been an ILP framework to produce a program from its interpretations. Learning from interpretations considers examples simply as single interpretations that are supposed to be models of an output program.

In the setting of [112, 113], it is assumed that each example is a small database (or a part of a global database), and local coverage tests are performed. This allows them to use algorithms based on local coverage tests that are usually linear in the number of examples. And, as each example can be loaded independently of the other ones, there is no need to use a database system even when the whole data set cannot be loaded into main memory.

When learning from interpretations, it is implicitly assumed that each example is completely specified. Indeed, in propositional logic, all propositions should be either true or false. As a consequence, missing values cannot be represented in this framework. It has therefore been suggested by [114] to represent examples by partial interpretations. In a partial interpretation, certain ground atoms have an unknown truth-value. Alternatively, [115] employs a second-order logic for dealing with this situation.

ProbLog [58, 116] is a probabilistic extension of Prolog. A ProbLog program defines a distribution over logic programs by specifying for each clause the probability that it belongs to a randomly sampled program, and these probabilities are mutually independent. In [117], the authors introduce a parameter learning algorithm from interpretations for ProbLog: LFI-ProbLog. LFI-ProbLog constructs a propositional logic formula for each interpretation that is used to estimate the marginals of the probabilistic parameters.

5.1.2 Learning Probabilistic Logic Program

Probabilistic inductive logic programming (or statistical relational learning) is the integration of probabilistic reasoning with first order logic representations and machine learning [118]. There is a lot of work lying at the intersection of probability theory, logic programming and machine learning [119–123]. This field of research is known under the names of statistical relational learning [124], probabilistic logic learning [125], or probabilistic inductive logic programming [118]. This field has already contributed a rich variety of valuable formalisms and techniques, including probabilistic Horn abduction [119], PRISMs [126], stochastic logic programs [122, 127, 128], Bayesian logic programs [129, 130], and Logical Hidden Markov Models [131].

5.1.3 Learning Action Theories

Abductive action learning has been studied based on *abductive event calculus* [132], an abductive extension of event calculus, and has been extended for applications to planning, e.g., [133]. Moyle [134] uses an ILP technique to learn a causal theory based on event calculus [49], given examples of input-output relations. In this work, a complete initial state is required as an input and a complete set of *narrative* facts is computed in advance. Otero [135] uses logic programs based on situation calculus [48], and considers causal theories represented in logic programs. In this work, the truth value of a goal fluent is assumed to change only once between two time points.

These previous works need either *frame axioms* or *inertia rules* in logic programs. The former causes the frame problem and the latter requires induction in nonmonotonic logic programs. Inoue *et al.* [136] induce causal theories represented in an action language given an incomplete action description and observations. But it requires an algorithm to learn finite automata to compute hypotheses, which may search the space of possible permutations of actions. Tran and Baral [137] define an action language which formalizes causal, trigger and inhibition rules to model signaling networks, and learn an action description in this language, given a candidate set of possible abducible rules. Active learning of action models is proposed by Rodrigues *et al.* [138] in a STRIPS-like language. Probabilistic logic programs to maximize the probabilities of observations are learned by Corapi *et al.* [139], by employing parameter estimation to find the probabilities associated with each atom and rule.

Works on learning action theories suppose applications to robotics and bioinformatics. In many action theories, one action is assumed to be performed at a time, so its learning task becomes sequential for each example sequence.

5.1.4 Reinforcement Learning

In some robotic systems, such as those that involve complex manipulation sequences, the decision maker has little input data and long periods of time to process it, because the actions take a long time to execute. A good approach for learning such tasks is *model based reinforcement learning* [140]. This approach allows a model to be obtained that represents the actions that the robot can execute. The model is generated from the experiences obtained when the robot executes the actions.

Work like [141], took profit of this representation to speed-up the learning process by combining relational reinforcement learning with active learning. *Active learning* is based on a free mix of exploration and instruction by an external teacher, and may be active in the sense that the system tests actions to maximize learning progress and asks the teacher if needed. Such approach is of

common use in robotic learning [142–145]. In [141], a relational model based on rules is learned and used to plan the best actions to do next. Furthermore, this model is also used to guide the teacher who helps the robot to learn the tasks.

5.1.5 Learning Nonmonotonic Programs

Learning NLPs has been considered in ILP, e.g., [146], but most approaches do not take the LFI setting. The LFI setting in learning NLPs is seen in Sakama [147]. Most approaches on learning NLPs is usually based on the stable model semantics [148].

The merit of the stable model semantics is that we can use state-of-the-art answer set solvers for computation of stable models. In [149], transition rules of Cellular Automata are represented in first-order NLPs where rules have a time argument. In this case, each NLP with the time argument becomes acyclic so the supported models and stable models coincide, and thus we can use answer set solvers for simulation of a CA. However, each answer set becomes infinite unless a time bound is set.

5.1.6 Learning Neural Networks

In [150], d’Avila Garcez and Zaverucha concentrate on the problem of extraction of symbolic knowledge from trained neural networks. The authors show that, for an NLP P , there exists a neural network N with bipolar semi-linear neurons that computes the immediate consequence operator T_P for P . This network N can also perform inductive learning from examples efficiently, assuming P as background knowledge and using the standard back-propagation learning algorithm. This method extracts knowledge directly from the internal part of the neural network. In [151], Lehmann et al. propose several algorithms to construct a logic program from a neural network. In this paper, they propose algorithms to construct an NLP from a mapping of interpretations of the input/output of a neural network.

In [152], the authors propose the Connectionist Inductive Learning and Logic Programming System ($C - IL^2P$). $C - IL^2P$ is a massively parallel computational model based on artificial neural networks. It integrates inductive learning from examples and background knowledge, with deductive learning from Logic Programming. Starting with the background knowledge represented by a propositional logic program, it applies a translation algorithm to generate a neural network that can be trained with examples. The authors propose to explain the results obtained by their refined network by extracting a revised logic program from it. Moreover, the neural network computes the stable model of the logic program inserted in it as background knowledge, or learned with the examples, thus functioning as a parallel system for Logic Programming. As a massively parallel non-monotonic learning system, $C - IL^2P$ has interesting implications for

the problem of Belief Revision. The background knowledge together with the set of examples can be inconsistent, and one needs to investigate ways to detect and treat inconsistencies in the system, viewing the learning process as a process of revision.

5.1.7 Learning Boolean Networks

Learning the dynamics of Boolean networks has been considered in Bioinformatics. Liang *et al.* [153] proposed the REVEAL algorithm, which uses mutual information in information theory as a measure of interrelationships. In REVEAL, the maximum number of arguments of each Boolean function is assumed to deal with exponential growth of computational time. Akutsu *et al.* [154] analyze the problem of identifying a genetic network from the data obtained by multiple gene disruptions and overexpressions with respect to the number of experiments. They show algorithms for identifying the underlying genetic network by such experiments, but their network model is a static Boolean network model in which expression levels of genes are statically determined, and is hence different from the standard Boolean network in which expression levels of genes change synchronously. Pal *et al.* [155] constructs Boolean networks from a partial description of state transitions. This method is considered as a method to complete missing transitions in the state transitions table. However, Boolean functions are not constructed for each node in [155].

In [156], Klarner *et al.* propose a method to find seeds and symbolic steady states [157] for model reduction and estimation of the number of cyclic attractors. Symbolic states are partial states and can be considered as a logic rule. The seeds are set of symbolic states and can be regarded as logic programs. The authors provide an optimization-based method for computing these seeds by exploiting the prime implicant graph of the Boolean network. This graph captures properties of fundamental importance for the network behavior. It permits to analyze certain aspects of asymptotic dynamics without having to calculate the state transitions graph.

Akutsu *et al.* [158] guess unknown Boolean functions of a Boolean network whose network topology is known. This corresponds to learning Boolean networks with the bias of neighbor nodes. In [158], only acyclic networks are considered, and the main focus is a computational analysis of such problems. Notably, all these previous works do not use ILP techniques.

Tamaddoni-Nezhad *et al.* [159] combine abduction and induction to learn rules of concentration changes of a metabolite caused by changes in other metabolites in a metabolic pathway. This method gives an empirical way to learn some causal effects, but its application domain does not deal with dynamical effects of feedbacks, and a learned program does not describe complete transitions of the dynamical system.

Inoue *et al.* [160] complete causal networks by meta-level abduction. A biological network can be constructed with this method for an incomplete structure, but the abductive method does not consider dynamical behavior of the network and cannot deal with negative feedbacks.

5.1.8 Learning Petri Nets

In [82], Srinivasan and Bain present a framework to learn Petri nets from state transitions. Petri nets can handle quantities of entities but their update schemes are different from those of Boolean networks. Here, a hierarchical Petri net can be obtained by iterative applications of their algorithm.

One of the domain of application of reinforcement learning is robotics. In some robotic systems, such as those that involve complex manipulation sequences, the decision maker has little input data and long periods of time to process it, because the actions take a long time to execute. A good approach for learning such tasks is *model based reinforcement learning* [140]. This approach allows a model to be obtained that represents the actions that the robot can execute. The model is generated from the experiences obtained when the robot executes the actions.

5.2 Comparison

In [50, 51], state transitions systems are represented with logic programs, 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) [42, 161].

Based on this ideas, we propose a framework to learn logic programs from traces of interpretation transitions: Learning From 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. As far as we know, this concept of *learning from interpretation transition* (LFIT) has never been considered in the ILP literature.

5.2.1 Computational learning theory

LFIT is different from any method to learn Boolean functions that has been developed in the field of computational learning theory [162]. **LFIT** learns dynamics of systems, while the conventional learning setting is not involved in dynamics. More generally, learning Boolean functions in the field of *computational learning theory* [162] is different from LFIT, since **LFIT** learns dynamics of systems as a set of Boolean functions appearing in Boolean networks, while the conventional learning setting is not involved in dynamics and often learns single Boolean functions. Similar to LFI, computational learning theories usually do not learn dynamics of systems in general.

5.2.2 Learning from interpretations

A closer setting can be found in *learning from interpretations* (LFI) [163] (Section 5.1.1), in which positive examples are given as Herbrand models of a target program, but again the goal of LFI is not to learn dynamics of systems. **LFIT** is different from the conventional LFI by De Raedt [163]. The setting for De Raedt's LFI learns a *clausal theory*, i.e., a set of clauses, instead of an NLP that is a set of rules of the form (2.1). A clause is simply a disjunction of literals, while a positive literal and a negative literal in the body are clearly distinguished in a rule of an NLP. Other than this syntactical difference, the algorithm of conventional LFI can be used to construct a clausal theory from our input. But, as stated in Section 5.1.1, LFI considers examples simply as single interpretations that are supposed to be models of an output program, hence is different from LFIT, which takes pairs of interpretations as its input. We actually see

that LFI is a special case of **LFIT**. That is, LFI can be constructed from **LFIT** as follows. Since $I \in 2^{\mathcal{B}}$ is a model of P iff $T_P(I) \subseteq I$, we can classify each example $(I, J) \in 2^{\mathcal{B}} \times 2^{\mathcal{B}}$ for **LFIT** into a positive example for LFI if $J \subseteq I$ or a negative example for LFI otherwise. The information of J is only used to check if I is a model or not in this conversion.

In [164], the authors investigate the issue of scaling up inductive logic programming within the setting of learning from interpretations. They propose two alternative implementations of the Tilde system [165]: Tildeclassic, which loads all data in main memory, and TildeLDS, which loads the examples one by one. Like TildeLDS, our firsts implementations of the **LFIT** framework learn iteratively by considering input examples one by one. But again, the goal of **LFIT** is to learn the dynamics of systems, which is not the case of the Tilde system.

In [117], the authors introduce a parameter learning algorithm from interpretations for ProbLog: LFI-ProbLog. LFI-ProbLog constructs a propositional logic formula for each interpretation that is used to estimate the marginals of the probabilistic parameters. We also extended our framework to learn probabilistic logic program (Section 4.5). But where LFI-ProbLog infer the probabilities of the composition of a state, **LFIT** infers the probabilities of the composition of a next state regarding the current state: the probabilities among dynamical properties; about transition. Typically, the rules outputed by the probabilistic learning algorithms of **LFIT** will provide the probability for a variable to take a specific value in the next state, according to the values of the other variable (in both current and next state).

5.2.3 Learning action theories

Learning action theories [134–139] (section 5.1.3) can be considered to share the common goals with **LFIT** on learning dynamics. The goal of learning action theories is not exactly the same as that of **LFIT**. In particular, **LFIT** can learn dynamics of systems with *positive and negative feedbacks*, which has not been considered much in the literature. In many action theories, one action is assumed to be performed at a time, so its learning task becomes sequential for each example sequence. For example, learning action theories by [135] assumes a sequence of actions in a narrative. In **LFIT**, on the other hand, every rule is fired as long as its body is satisfied and update is synchronously performed at every ground atom. Moyle [134] uses an ILP technique to learn a causal theory based on event calculus [49], given examples of input-output relations. But in this work, a complete initial state is required as an input and a complete set of *narrative* facts is computed in advance, and thus observations handled in our work cannot be explained.

5.2.4 Reinforcement learning

As discussed in section 5.1.4, one of the domain of application of reinforcement learning is robotics. A good approach for learning robot tasks is relational reinforcement learning. In [141], where a relational model based reinforcement learning approach is presented, the model used is based on rules that are learned and used to plan the robot actions as well as to guide the teacher who helps the robot to learn the tasks. In such approach, **LFIT** methods can be used by to learn this model.

In Džeroski *et al.* [166]’s *relational reinforcement learning* (RRL) is a learning technique that combines reinforcement learning with ILP. As in (non-relational) reinforcement learning, RRL can take into account feedbacks from the learning process as rewards: each time an observation is received, an action is chosen so that the state is changed with the reward associated. The goal of RRL is then to find a suitable sequence of transitions that maximize rewards. The merit to use ILP in RRL is to have a more expressive representation in states, actions and Q-functions. With a relational representation, the states, actions, and transitions are not represented individually. Entities of the same predefined type are grouped and their relationships are considered. The generalization provided by those models reduce the learning complexity. Contrary to [166], that can consider feedbacks in the learning process as rewards, **LFIT** learns how such feedbacks can be represented logically by state transitions rules. As the motivation of RRL is different from that of LFIT, our goal is not to find an optimal strategy for state transitions but to learn the system’s dynamics itself. As for the treatment of positive and negative feedbacks, **LFIT** learns how such feedbacks can be represented by logic programs.

5.2.5 Learning Nonmonotonic Programs

Learning NLPs rather than definite programs has been considered in ILP, e.g., [146], but most approaches do not take the LFI setting. Our learning framework is different from these previous works [146, 147]. From the application point of view, NLPs are often used in planning and robotics domain. Hence, the difference between previous work on learning action theories and **LFIT** is inherited to the comparison between previous setting of learning NLPs and **LFIT**. From the semantical viewpoint, there is an additional important difference: previous work on learning NLPs is usually based on the stable model semantics [148], but **LFIT** learns NLPs under the supported model (or supported set) semantics [51]. The merit of the supported model semantics is that we can omit the time argument from a program and make it simpler. Boolean networks can be represented in propositional NLPs [50], but still we can simulate state transitions by watching the orbits of the T_P operator. More importantly, attractors can be directly obtained with the supported model or the supported set semantics. This is not possible using the stable models of NLPs (without the time argument), since they ignore all positive feedback loops in the

dynamics [50]. The supported models of an NLP can also be obtained as the models of Clark's completion of the program using modern SAT solvers. If we use answer set solvers for an NLP with the time argument, we can simulate the dynamics of the corresponding Boolean networks, but need to analyze each answer set to know when the same state is encountered twice by tracing the orbit from time to time.

Learning with Neural Networks:

In [150], the authors propose a method to extract symbolic knowledge from trained neural networks. Given a set of input/output of a neural network, **LFIT** could be used to learn a logic program that capture the dynamic of the neural network. In [150], knowledge is extracted from the topology (inside) of the system where **LFIT** learn from its behavior (outside). This method suppose that the internal part of the neural network is accessible.

In [152], the authors propose the Connectionist Inductive Learning and Logic Programming System. $C - IL^2P$ is a massively parallel computational model based on artificial neural networks. The authors propose to explain the results obtained by their refined network by extracting a revised logic program from it. Moreover, the neural network computes the stable model of the logic program inserted in it as background knowledge, or learned with the examples, thus functioning as a parallel system for Logic Programming. **LFIT** can also start with a logic program as background knowledge and performs inductive learning from trace of state transitions that can be seen as training examples. But, the background knowledge together with the set of examples can be inconsistent, and one needs to investigate ways to detect and treat inconsistencies in the system, viewing the learning process as a process of revision. **LFIT** share the same concerns, to successfully learn a Boolean network from its states transition, **LFIT** can only infer consistent rules, i.e the state transitions have to be consistent.

In [151], Lehmann et al. propose several algorithms to construct a logic program from a neural network. Here, they propose algorithms to construct an NLP from a mapping of interpretations, whose goal is similar to that of **LFIT**. Although there are some differences, one of their reduction method, called q-subsumption, corresponds to the ground resolution (Definition 3.1) of **LFIT**. The main difference is that **LFIT** reduces the NLP iteratively, while in [151] all interpretations are analyzed in a batch. Then, a large program cannot be handled in such a reduction method, and the iterative reduction done by **LFIT** will be much more efficient in both memory use and run time. The first algorithm they propose infer rules from interpretation transition in the same maner as **LFIT** with generalization does. All infered rules are stored in one set S and all generalisation of the rule of S are computed. This method will always be slower than **LFIT**, since such an extrem case cannot append in our algorithms: all generalizations are made each time we learn a new rule, thus, the program learn cannot be that big. The authors also provide a greedy algorithm to perform the same task. This method iterate on all possible rules, starting by the most general ones it keep the ones that are consistent with the set of interpretations. It

can seem pretty similar to our method based on minimal specialization. But again, many more rules will be generated with their method and these rules will be compared with much more transitions. Beside the performance issue and the non-iterative character of this method, it is the most related work.

5.2.6 Learning Probabilistic Logic Program

We extended the *LFIT* framework to learn probabilistic dynamics by proposing an extension of *LFIT* for learning from uncertain state transitions. Where work like [106] perform inferences from a probabilistic logic program, what we do is inferring the rules of such logic program. The programs inferred by our new algorithm is similar to paraconsistent logic program of [99]. The use of annotated atoms allows the programs learned to induce multiple values for the same represented variable. It allows us to represent multi-valued model and capture non-deterministic state transitions. Our semantic differs from previous work like [107]. Here, the authors consider probabilistic logic programs as logic programs in which some of the facts are annotated with probabilities. But in our method, it's the rules that have probabilities and they are independent.

5.2.7 Learning Boolean Networks

An intended direct application of **LFIT** is learning transition or update rules in *dynamical systems* such as *Boolean networks* [75] and *cellular automata* [76], which have been respectively used as mathematical models of genetic networks and complex adaptive systems. It has been observed that the T_P operator for an NLP P precisely captures the synchronous update of the corresponding Boolean network, where each gene and its regulation function correspond to a ground atom and the set of ground rules with the atom in their heads, respectively [50]. Then, given an input Herbrand interpretation I , which corresponds to a *gene activity profile* (GAP) with gene disruptions for false atoms in I and gene overexpressions for true atoms in I , the interactions between genes are experimentally analyzed by observing an output GAP J such that $J = T_P(I)$ is assumed to hold after a time step has passed. In this setting, **LFIT** of an NLP P corresponds to inferring a set of gene regulation rules that are complete for those experiments of 1-step GAP transitions. Such a learning task has been analyzed in the literature [154, 158], but no ILP technique has been applied to the problem. Besides, 2-state cellular automata, in which each cell can take either 1 or 0 as a possible value, are instances of Boolean networks, so that their state transitions are determined by the T_P operator [167]. Hence it is possible to apply **LFIT** for their learning tasks. Learning transition rules (called *identification*) of cellular automata has been studied in the literature [84, 168], but again no previous work has employed ILP techniques on this problem.

Learning the dynamics of Boolean networks has been considered in Bioinformatics [153–155] as we discussed in section 5.1.7. Compared with these studies, our learning method is a complete algorithm to learn a set of logical state transitions rules for a Boolean network. As in [155], we can also deal with partial transitions, but will not need to identify or enumerate all possible complete transitions.

In [156], Klarner *et al.* provide an optimization-based method for computing model reduction by exploiting the prime implicant graph of the Boolean network. The prime implicant graph is similar to the prime implicant rules that can be learned by **LFIT** (section 3.3). But where [156] work directly on the model, **LFIT** learn from the transitions of the model. Furthermore, to compute model reduction the methods of [156] needs to enumerate all prime implicants. The number of prime implicants of a Boolean formula grows exponentially with the number of variables it depends on. When learning prime implicants rules, **LFIT** will only infer the rules needed to capture the network behavior. But, according to [156], for typical biological models these dependencies are so small that this enumeration is negligible. In our case, the original purpose of **LFIT** is to learn system, like Boolean networks, whose model is unknown. Here, the dependencies among variables are unknown and the enumeration of all prime implicants is no more negligible. But our inference methods could also be used for the simplification of existing models. Knowing the model, one could generate its state transitions and then use **LFIT** to learn a reduced model. Knowing the influences among the variables of the system, we could generate partial state transitions to directly learn general rules. But when the model is already known, it should be more efficient to just use a method that work at the model level.

In [45], Lähdesmäki *et al.* propose algorithms to infer the Boolean functions of gene regulatory network from gene expression data. Here, gene expression data is given as a set of positive and negative example for each Boolean function. Each positive (resp. negative) example represents a variable configuration that makes a Boolean function true (resp. false). The authors propose an algorithm to construct the truth table of a Boolean function according to this set of examples. This is pretty similar to learning from interpretation of transitions: for a transition (I, J) , for every A in J a positive example of the Boolean function of the variable A is I and for every A' in the Herbrand base that are not in J , I is a negative example of the Boolean function of the variable A' . The purpose of the algorithm proposed in [45] is also to check the consistency of the given gene expression data. The author also extended their method to learn gene regulatory networks under the Best-Fit Extension paradigm. The Best-Fit Extension Problem is to find a Boolean functions that do as few “misclassification” as possible, on a given set of positive and negative examples. Most algorithm in our framework assume that transitions are consistent, i.e. from a given state there is exactly one next state. **LFIT** cannot be used to directly find the most likely Boolean function. But, one could compute this function from the output of the algorithm we propose to deal with uncertain state transitions in section 4.5.

5.2.8 Learning Petri Nets

In [82], a hierarchical Petri net can be obtained by iterative applications of their algorithm. but it is not possible to obtain networks with positive and negative feedback cycles. In fact, cyclic dependencies have been generally hard to learned in ILP methods. Similarly to this work, the main idea of the **LFIT** framework is to infer a system from the observations of its transitions. One of our algorithm can capture the delayed influences of systems, this algorithm could be used to learn Bounded Petri Nets. Using this algorithm, we could learn positive and negative feedback cycles.

5.2.9 Learning Cellular Automata

In cellular automata (CAs), constructing transition rules from given configurations is known as the *identification problem*. Adamatzky [84] provides algorithms for identifying different classes of CAs, and analyzes computational complexities of those algorithms. Several algorithms are also proposed in [168]. To the best of our knowledge, however, there is no algorithm which uses ILP techniques for identifying CA rules.

5.2.10 Binary Decision Diagram

The probabilistic logic programming language ProbLog [58] computes probabilities via Binary Decision Diagrams. A ProbLog program computes the probability of a query atom by applying sum-product computation to a Binary Decision Diagram, but allows definite clauses only. For abduction in propositional theories, Simon and del Val [59] propose a consequence-finding procedure implemented on Zero-suppressed BDDs. [60] run the EM algorithm over BDDs to evaluate abductive hypotheses. In one of our implementations of the resolution based version of LFIT, we took inspiration from Binary Decision Diagram for our data structure that represent the logic rules learned. The compact representation of Binary Decision Diagram allows to drastically increase the performances of **LFIT** in terms of both runtime and memory.

5.2.11 Inverse Ingeniering

In [169], Pellegrino and Balzarotti propose a black-box technique to detect logic vulnerabilities in web applications. Their approach is based on the automatic identification of a number of behavioral patterns. Starting from few network traces in which users interact with a certain application. Based on the extracted model, they generate targeted test cases following a number of common attack scenarios.

In practice, the most that can do the current **LFIT** implementation is to learn a single webpage form. Where the possible values of each field has to be known, i.e. checkbox/choice list. On such simple web form, **LFIT** could infer the model behind the webpage, but will not provide the potential weak point directly. What **LFIT** does can be seen like inverse engineering, it could be used to automatically reconstruct a model of an application or of its protocol like in [170, 171].

Chapter 6

Conclusions and Future Work

This thesis investigates the studies of the automatic construction of model of the dynamics of a system from the observation of its state transitions. Given some raw data on the process, like time series data of gene expression, we assume a discretization of those data in the form of state transitions. From those state transitions, according to the semantic of the system dynamics, we propose different inference algorithms that model the system as a logic program. The semantic of system dynamics can differ regarding the synchronicity of its variables, the determinism of its changes and the influence of its history. In this thesis we propose several modelings and learning algorithms to tackle those different semantics.

6.1 Summary of contribution

We proposed several algorithms for learning Boolean synchronous deterministic system from interpretation transitions. Given any state transitions diagram we can now learn an NLP that exactly captures the system dynamics. Consistency of state transitions rules is achieved and minimality of rules is guaranteed. As a result, given any state transitions diagram E , **LF1T** with least specialization always learns a unique NLP that contains all prime rules that realize E . It implies that the output of **LF1T** is no more sensitive to variable ordering or transition ordering. But, experimental results showed that the new algorithm is sensitive to input transitions ordering regarding run time. Design of an heuristic to make a good ordering of the input is one possible future work.

To understand the memory effect involved in some interactions between biological components, it is necessary to include delayed influences in the model. In this thesis, we proposed a logical method to learn such models from state transitions systems. We designed an approach to learn Boolean networks with delayed influences. This can be directly applied to the learning of

Boolean (or multi-level 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 [91]. 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. In account of the noise inherent to biological data, the ability to either perform an efficient discretization of the data or to include the notion of noise inside the modeling framework is fundamental. We will thus have to discuss the discretization procedure and the robustness of our modeling against noisy data and compare it to existing approaches, like the Bayesian ones [93].

Finally, we have presented a new approach to learn planning operators. In contrast to previous approaches, it learns the dynamics of every predicate independently, and thus can be applied to learn RDDDL domains. The presented approach can find good sets of planning operators efficiently. As future work, we would like to work on planning operators independent of actions. This would require to have a more intelligent method to generate symbolic transitions from grounded transitions so that changes not represented by actions could be learned. Our probabilistic approach can be used to deal with the problems caused by noisy data as discussed in previous paragraph. Indeed, if the quantity of input data is sufficiently big the probabilities of the rules should discriminate between the real dynamic, i.e. the rules with high probabilities, and the rules produced by the noise, i.e. the rules with low probabilities.

6.2 Perspectives

One possible use of our method is the construction and revision of model to fit some required properties on the dynamics. The idea is to infer rules consistent with temporal properties or constraints so that the model represented by the learned logic program basically does what we want. A naïve approach to this problem is to simply generate state transitions diagrams until we find one that is consistent with all properties. Then our framework can be used to learn the corresponding model of the selected diagram. This first method was in a collaborative work with Alexandre Rocca, published as a book chapter in *Logical Modeling of Biological Systems* [172]. A more interesting approach would be to directly construct rules from the properties we want to be satisfied. For some simple temporal properties (e.g., a given variable is expected to have a particular value in the next state of the model), this is straightforward. But for other properties that require exploration of the state transitions diagram, this is not so trivial. An elegant way could be to start with a logic program, either given or eventually composed of all fact rules. Here, we could simply run this logic program to capture the transitions that are not consistent with the required temporal properties. Then we could use our specialization method to remove those transitions from the dynamic of the program. This could ensure that the dynamics of the

learned system does not violate a constraint. But by doing so, some other properties may not hold anymore. The solution could be to use a mix of specialization and generalization: specialize to avoid undesired transitions and generalize to satisfy desired transitions. This is a very challenging problem, because depending of the properties we want to achieve: the approach could require to run the learned program so many times that we will generate more transitions than by just computing the whole state transitions diagram once. So far, it seems more reasonable in terms of run time efficiency to simply build something like a partial state transitions diagram. It would be a more general representation than the whole diagram but where the properties could still be checked. From this partial diagram, it will then be straightforward to construct the model using our framework. Construction of models that fit temporal properties is an interesting problem and it is a challenging topic in the field of systems biology. Usually, biologist know some properties of the system they try to model. Being able to use those properties during the learning of a model from time series data by our framework could make it more usable in practice and more efficient.

Another interesting possible application of our framework is data compilation. There is a lot of efficient compilation methods that exists for text files, but nothing really efficient in terms of compilation for binary files. Thus, the method we propose to learn delayed influences could be used for this purpose. The idea is to learn a more compact representation of these files in the form of a markov(k) system. By arbitrary choosing a number n of bits, we can consider a binary file as a unique sequence of state transitions where each state is n bits values. Then, our algorithm can learn a logic program that represents a Markov(k) system which exactly realize this sequence. Thus, this logic program could be more a compact representation of the binary file. So far, the current implementation of the algorithm is not efficient for this kind of task, both regarding run time and compression rate. The computation of the minimal delay alone is already costly and the choosing a good number of bit to consider as a state is a complex problem itself. One idea could be to divide the file in multiple parts where in each of them the transitions can be explained with a small delay. Further work is needed to make this idea usable in practice, this may be the occasion to investigate another application field.

Bibliography

- [1] Jean-Paul Comet, Jonathan Fromentin, Gilles Bernot, and Olivier Roux. A formal model for gene regulatory networks with time delays. In Computational Systems-Biology and Bioinformatics, pages 1–13. Springer, 2010.
- [2] Katsumi Inoue, Tony Ribeiro, and Chiaki Sakama. Learning from interpretation transition. Machine Learning, 94(1):51–79, 2014.
- [3] A.L. Samuel. Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3):210–229, July 1959. ISSN 0018-8646. doi: 10.1147/rd.33.0210.
- [4] Tom M Mitchell. Machine learning and data mining. Communications of the ACM, 42(11):30–36, 1999.
- [5] Alan M Turing. Computing machinery and intelligence. Mind, pages 433–460, 1950.
- [6] Stevan Harnad. The annotation game: On turing (1950) on computing, machinery, and intelligence. The Turing Test Sourcebook: Philosophical and Methodological Issues in the Quest for the Thinking Computer, 2006.
- [7] Andrew G Barto. Reinforcement learning: An introduction. MIT press, 1998.
- [8] Jacques Herbrand. Recherches sur la théorie de la démonstration. 1930. URL <http://eudml.org/doc/192791>.
- [9] Willard Van Orman Quine. Logic and the reification of universals. From a logical point of view, 6, 1953.
- [10] John McCarthy. An algebraic language for the manipulation of symbolic expressions, 1958.
- [11] John Alan Robinson. A machine-oriented logic based on the resolution principle. Journal of the ACM (JACM), 12(1):23–41, 1965.
- [12] Robert A Kowalski. The early years of logic programming. Communications of the ACM, 31(1):38–43, 1988.

- [13] EW Elcock. Absys: The first logic programming language—a retrospective and a commentary. The Journal of Logic Programming, 9(1):1–17, 1990.
- [14] Carl Hewitt. Planner: A language for proving theorems in robots. In IJCAI, pages 295–302, 1969.
- [15] Gerald Sussman and Terry Winograd. Micro-planner reference manual. 1970.
- [16] Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, DTIC Document, 1971.
- [17] Johns F Rulifson, Jan A Derksen, and Richard J Waldinger. Qa4: A procedural calculus for intuitive reasoning. Technical report, DTIC Document, 1972.
- [18] Drew V McDermott and Gerald Jay Sussman. The conniver reference manual. 1972.
- [19] Julian M Davies. Popler 1.5 reference manual. Theoretical Psychology Unit, School of Artificial Intelligence, University of Edinburgh, 1973.
- [20] Earl D Sacerdoti, Richard E Fikes, Rene Reboh, Daniel Sagalowicz, Richard J Waldinger, and B Michael Wilber. Qlisp: a language for the interactive development of complex systems. In Proceedings of the June 7-10, 1976, national computer conference and exposition, pages 349–356. ACM, 1976.
- [21] William A Kornfeld and Carl E Hewitt. The scientific community metaphor. Systems, Man and Cybernetics, IEEE Transactions on, 11(1):24–33, 1981.
- [22] Patrick J Hayes. Computation and deduction. 1972.
- [23] A Colmeraner, Henri Kanoui, Robert Pasero, and Philippe Roussel. Un systeme de communication homme-machine en francais. Luminy, 1973.
- [24] Nils J Nilsson. Probabilistic logic. Artificial intelligence, 28(1):71–87, 1986.
- [25] Arthur P Dempster. Upper and lower probabilities induced by a multivalued mapping. The annals of mathematical statistics, pages 325–339, 1967.
- [26] Alan R Anderson, Nuel D Belnap, et al. Entailment, vol. 1. Princeton UP Princeton, 1975.
- [27] Audun Jøsang. A logic for uncertain probabilities. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 9(03):279–311, 2001.
- [28] Audun Jøsang and David McAnally. Multiplication and comultiplication of beliefs. International Journal of Approximate Reasoning, 38(1):19–51, 2005.

- [29] Audun Josang. Conditional reasoning with subjective logic. Journal of Multiple-Valued Logic and Soft Computing, 15(1):5–38, 2008.
- [30] George Klir and Bo Yuan. Fuzzy sets and fuzzy logic, volume 4. Prentice Hall New Jersey, 1995.
- [31] Giangiacomo Gerla. Inferences in probability logic. Artificial Intelligence, 70(1):33–52, 1994.
- [32] Matthew Richardson and Pedro Domingos. Markov logic networks. Machine learning, 62(1-2):107–136, 2006.
- [33] Cornelis J Van Rijsbergen. A non-classical logic for information retrieval. The computer journal, 29(6):481–485, 1986.
- [34] Edwin T Jaynes. Information theory and statistical mechanics. Physical review, 106(4): 620, 1957.
- [35] Edwin T Jaynes. Information theory and statistical mechanics. ii. Physical review, 108 (2):171, 1957.
- [36] Pei Wang. Non-Axiomatic Reasoning System— Exploring the Essence of Intelligence. PhD thesis, Citeseer, 1995.
- [37] Ben Goertzel, Matthew Iklé, Izabela Freire Goertzel, and Ari Heljakka. Probabilistic logic networks: A comprehensive framework for uncertain inference. Springer Science & Business Media, 2008.
- [38] Enrique H Ruspini, John D Lowrance, and Thomas M Strat. Understanding evidential reasoning. International Journal of Approximate Reasoning, 6(3):401–424, 1992.
- [39] Stephen Muggleton. Inductive logic programming. New generation computing, 8(4): 295–318, 1991.
- [40] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. The Journal of Logic Programming, 19–20, Supplement 1(0):629 – 679, 1994. ISSN 0743-1066. doi: [http://dx.doi.org/10.1016/0743-1066\(94\)90035-3](http://dx.doi.org/10.1016/0743-1066(94)90035-3). URL <http://www.sciencedirect.com/science/article/pii/0743106694900353>. Special Issue: Ten Years of Logic Programming.
- [41] J. Ross Quinlan. Induction of decision trees. Machine learning, 1(1):81–106, 1986.
- [42] Krzysztof R Apt, Howard A Blair, and Adrian Walker. Towards a theory of declarative knowledge. Foundations of deductive databases and logic programming, page 89, 1988.

- [43] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. The Journal of Logic Programming, 19:629–679, 1994.
- [44] S. A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. Journal of theoretical biology, 22(3):437–467, 1969.
- [45] Harri Lähdesmäki, Ilya Shmulevich, and Olli Yli-Harja. On learning gene regulatory networks under the boolean network model. Machine Learning, 52(1-2):147–167, 2003.
- [46] S. Klamt, J. Saez-Rodriguez, J. A. Lindquist, L. Simeoni, and E. Dieter Gilles. A methodology for the structural and functional analysis of signaling and regulatory networks. BMC Bioinformatics, 7:56, 2006.
- [47] Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter Flach, Katsumi Inoue, and Ashwin Srinivasan. Ilp turns 20. Machine Learning, 86(1):3–23, 2012.
- [48] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. Stanford University USA, 1968.
- [49] Robert Kowalski and Marek Sergot. A logic-based calculus of events. In Foundations of knowledge base management, pages 23–55. Springer, 1989.
- [50] Katsumi Inoue. Logic programming for boolean networks. In Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two, pages 924–930. AAAI Press, 2011.
- [51] Katsumi Inoue and Chiaki Sakama. Oscillating behavior of logic programs. In Correct Reasoning, pages 345–362. Springer, 2012.
- [52] Elisabeth Remy and Paul Ruet. From minimal signed circuits to the dynamics of boolean regulatory networks. Bioinformatics, 24(16):i220–i226, 2008.
- [53] Tony Ribeiro, Katsumi Inoue, and Chiaki Sakama. A bdd-based algorithm for learning from interpretation transition. In Inductive Logic Programming, pages 47–63. Springer, 2014.
- [54] Sheldon B. Akers. Binary decision diagrams. Computers, IEEE Transactions on, 100(6):509–516, 1978.
- [55] Randal E Bryant. Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on, 100(8):677–691, 1986.
- [56] Fadi A Aloul, Maher N Mneimneh, and Karem A Sakallah. Zbdd-based backtrack search sat solver. In Proc. Int’l Workshop on Logic Synthesis, Lake Tahoe, California, 2002.

- [57] Shinichi Minato and Hiroki Arimura. Frequent closed item set mining based on zero-suppressed bdds. Information and Media Technologies, 2(1):309–316, 2007.
- [58] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In IJCAI, volume 7, pages 2462–2467, 2007.
- [59] Laurent Simon and Alvaro Del Val. Efficient consequence finding. In International Joint Conference on Artificial Intelligence, volume 17, pages 359–370. LAWRENCE ERLBAUM ASSOCIATES LTD, 2001.
- [60] Katsumi Inoue, Taisuke Sato, Masakazu Ishihata, Yoshitaka Kameya, and Hidetomo Nabeshima. Evaluating abductive hypotheses using an em algorithm on bdds. In Proceedings of the 21st international joint conference on Artificial intelligence, pages 810–815. Morgan Kaufmann Publishers Inc., 2009.
- [61] Randal E Bryant and Christoph Meinel. Ordered binary decision diagrams. Springer, 2002.
- [62] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR), 24(3):293–318, 1992.
- [63] Shinichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In 30th Conference on Design Automation, pages 272–277. IEEE, 1993.
- [64] Tony Ribeiro and Katsumi 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>).
- [65] Ryszard S Michalski. A theory and methodology of inductive learning. Artificial intelligence, 20(2):111–161, 1983.
- [66] Tom M Mitchell. Generalization as search. Artificial intelligence, 18(2):203–226, 1982.
- [67] Katsumi Inoue. Dnf hypotheses in explanatory induction. In Inductive Logic Programming, pages 173–188. Springer, 2012.
- [68] Pierre Tison. Generalization of consensus theory and application to the minimization of boolean functions. Electronic Computers, IEEE Transactions on, (4):446–456, 1967.
- [69] Alex Kean and George Tsiknis. An incremental method for generating prime implicants/implicates. Journal of Symbolic Computation, 9(2):185–206, 1990.
- [70] Jean-Paul Comet, Gilles Bernot, Aparna Das, Francine Diener, Camille Massot, and Amélie Cessieux. Simplified models for the mammalian circadian clock. Procedia Computer Science, 11:127–138, 2012.

- [71] Wassim Abou-Jaoudé, Djomangan A Ouattara, and Marcelle Kaufman. From structure to dynamics: frequency tuning in the p53–mdm2 network: I. logical approach. Journal of theoretical biology, 258(4):561–577, 2009.
- [72] Tatsuya Akutsu, Satoru Kuhara, Osamu Maruyama, and Satoru Miyano. Identification of genetic networks by strategic gene disruptions and gene overexpressions under a boolean model. Theoretical Computer Science, 298(1):235–251, 2003.
- [73] Tony Ribeiro, Morgan Magnin, Katsumi Inoue, and Chiaki Sakama. Learning delayed influences of biological systems. Frontiers in Bioengineering and Biotechnology, 2:81, 2015.
- [74] John Alan Robinson. A machine-oriented logic based on the resolution principle. Journal of the ACM (JACM), 12(1):23–41, 1965.
- [75] Stuart A. Kauffman. The origins of order: Self-organization and selection in evolution. Oxford university press, 1993.
- [76] Stephen Wolfram. Cellular automata and complexity: collected papers, volume 152. Addison-Wesley Reading, 1994.
- [77] Elena Dubrova and Maxim Teslenko. A sat-based algorithm for finding attractors in synchronous boolean networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), 8(5):1393–1399, 2011.
- [78] Alvaro Chaos, Max Aldana, Carlos Espinosa-Soto, Berenice García Ponce de León, Adriana Garay Arroyo, and Elena R Alvarez-Buylla. From genes to flower patterns and evolution: dynamic models of gene regulatory networks. Journal of Plant Growth Regulation, 25(4):278–289, 2006.
- [79] Fangting Li, Tao Long, Ying Lu, Qi Ouyang, and Chao Tang. The yeast cell-cycle network is robustly designed. Proceedings of the National Academy of Sciences of the United States of America, 101(14):4781–4786, 2004.
- [80] Maria I Davidich and Stefan Bornholdt. Boolean network model predicts cell cycle sequence of fission yeast. PloS one, 3(2):e1672, 2008.
- [81] Adrien Fauré, Aurélien Naldi, Claudine Chaouiya, and Denis Thieffry. Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle. Bioinformatics, 22(14):e124–e131, 2006.
- [82] Ashwin Srinivasan and Michael Bain. Knowledge-guided identification of petri net models of large biological systems. In Inductive Logic Programming, pages 317–331. Springer, 2012.

- [83] Luis Mendoza and Ioannis Xenarios. Theoretical biology and medical modelling. Theoretical Biology and Medical Modelling, 3:13, 2006.
- [84] Andrew I Adamatzky. Identification of cellular automata. CRC Press, 1994.
- [85] Heh-Tyan Liaw and Chen-Shang Lin. On the obdd-representation of general boolean functions. IEEE Trans. Computers, 41(6):661–664, 1992.
- [86] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.
- [87] Maya Paczuski, Kevin E. Bassler, and Álvaro Corral. Self-organized networks of competing boolean agents. Phys. Rev. Lett., 84:3185–3188, Apr 2000. doi: 10.1103/PhysRevLett.84.3185. URL <http://link.aps.org/doi/10.1103/PhysRevLett.84.3185>.
- [88] Adrian Silvescu and Vasant Honavar. Temporal boolean network models of genetic networks and their inference from gene expression time series. Complex Systems, 13(1): 61–78, 2001.
- [89] Paul T Spellman, Gavin Sherlock, Michael Q Zhang, Vishwanath R Iyer, Kirk Anders, Michael B Eisen, Patrick O Brown, David Botstein, and Bruce Futcher. Comprehensive identification of cell cycle-regulated genes of the yeast *saccharomyces cerevisiae* by microarray hybridization. Molecular biology of the cell, 9(12):3273–3297, 1998.
- [90] Raymond J Cho, Michael J Campbell, Elizabeth A Winzeler, Lars Steinmetz, Andrew Conway, Lisa Wodicka, Tyra G Wolfsberg, Andrei E Gabrielian, David Landsman, David J Lockhart, et al. A genome-wide transcriptional analysis of the mitotic cell cycle. Molecular cell, 2(1):65–73, 1998.
- [91] Xia Li, Shaoqi Rao, Wei Jiang, Chuanxing Li, Yun Xiao, Zheng Guo, Qingpu Zhang, Lihong Wang, Lei Du, Jing Li, et al. Discovery of time-delayed gene regulatory networks based on temporal gene expression profiling. BMC bioinformatics, 7(1):26, 2006.
- [92] Gerhard Brewka, Thomas Eiter, and Miros Truszczyński. Answer set programming at a glance. Commun. ACM, 54(12):92–103, December 2011. ISSN 0001-0782. doi: 10.1145/2043174.2043195. URL <http://doi.acm.org/10.1145/2043174.2043195>.
- [93] Nathan A Barker, Chris J Myers, and Hiroyuki Kuwahara. Learning genetic regulatory network connectivity from time series data. Computational Biology and Bioinformatics, IEEE/ACM Transactions on, 8(1):152–165, 2011.

- [94] Michael Kifer and VS Subrahmanian. Theory of generalized annotated logic programming and its applications. Journal of Logic Programming, 12(4):335–367, 1992.
- [95] Melvin Fitting. Bilattices and the semantics of logic programming. The Journal of Logic Programming, 11(2):91 – 116, 1991. ISSN 0743-1066. doi: [http://dx.doi.org/10.1016/0743-1066\(91\)90014-G](http://dx.doi.org/10.1016/0743-1066(91)90014-G). URL <http://www.sciencedirect.com/science/article/pii/074310669190014G>.
- [96] Matthew L Ginsberg. Multivalued logics: A uniform approach to reasoning in artificial intelligence. Computational intelligence, 4(3):265–316, 1988.
- [97] Maarten H Van Emden. Quantitative deduction and its fixpoint theory. The Journal of Logic Programming, 3(1):37–53, 1986.
- [98] Howard A. Blair and V.S. Subrahmanian. Paraconsistent logic programming. Theoretical Computer Science, 68(2):135 – 154, 1989. ISSN 0304-3975. doi: [http://dx.doi.org/10.1016/0304-3975\(89\)90126-6](http://dx.doi.org/10.1016/0304-3975(89)90126-6). URL <http://www.sciencedirect.com/science/article/pii/0304397589901266>.
- [99] Howard A Blair and VS Subrahmanian. Paraconsistent foundations for logic programming. Journal of non-classical logic, 5(2):45–73, 1988.
- [100] Abhishek Garg, Alessandro Di Cara, Ioannis Xenarios, Luis Mendoza, and Giovanni De Micheli. Synchronous versus asynchronous modeling of gene regulatory networks. Bioinformatics, 24(17):1917–1925, 2008.
- [101] Avraham A Melkman, Takeyuki Tamura, and Tatsuya Akutsu. Determining a singleton attractor of an and/or boolean network in $O(n \log n)$ time. Information Processing Letters, 110(14):565–569, 2010.
- [102] Tatsuya Akutsu, Avraham A Melkman, Takeyuki Tamura, and Masaki Yamamoto. Determining a singleton attractor of a boolean network with nested canalizing functions. Journal of Computational Biology, 18(10):1275–1290, 2011.
- [103] Gilles Bernot, Jean-Paul Comet, Adrien Richard, and Janine Guespin. Application of formal methods to biological regulatory networks: extending thomas bi asynchronous logical approach with temporal logic. Journal of theoretical biology, 229(3):339–347, 2004.
- [104] Adrien Fauré, Aurélien Naldi, Claudine Chaouiya, and Denis Thieffry. Dynamical analysis of a generic boolean model for the control of the mammalian cell cycle. Bioinformatics, 22(14):e124–e131, 2006.

- [105] René Thomas and Marcelle Kaufman. Multistationarity, the basis of cell differentiation and memory. ii. logical analysis of regulatory networks in terms of feedback circuits. Chaos: An Interdisciplinary Journal of Nonlinear Science, 11(1):180–195, 2001.
- [106] Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. Theory and Practice of Logic Programming, 11(4-5):663–680, 2011.
- [107] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory and Practice of Logic Programming, pages 1–44, 2013.
- [108] Håkan LS Younes and Michael L Littman. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. Techn. Rep. CMU-CS-04-162, 2004.
- [109] Scott Sanner. Relational dynamic influence diagram language (RDDL): Language description. Unpublished ms. Australian National University, 2010.
- [110] Leslie G Valiant. A theory of the learnable. Communications of the ACM, 27(11):1134–1142, 1984.
- [111] Dana Angluin, Michael Frazier, and Leonard Pitt. Learning conjunctions of horn clauses. Machine Learning, 9(2-3):147–164, 1992.
- [112] Luc De Raedt and Sašo Džeroski. First-order jk-clausal theories are pac-learnable. Artificial Intelligence, 70(1):375–392, 1994.
- [113] Luc De Raedt. Attribute-value learning versus inductive logic programming: The missing links. In Inductive Logic Programming, pages 1–8. Springer, 1998.
- [114] Dieter Fensel, Monika Zickwolff, and Markus Wiese. Are substitutions the better examples? learning complete sets of clauses with frog. In Proceedings of the 5th International Workshop on Inductive Logic Programming, pages 453–474. Citeseer, 1995.
- [115] Peter A Flach. A framework for inductive logic programming. Inductive Logic Programming, 38:193–211, 1992.
- [116] Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. Parameter learning in probabilistic databases: A least squares approach. In Machine Learning and Knowledge Discovery in Databases, pages 473–488. Springer, 2008.
- [117] Bernd Gutmann, Ingo Thon, and Luc De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In Machine Learning and Knowledge Discovery in Databases, pages 581–596. Springer, 2011.

- [118] Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In Algorithmic Learning Theory, pages 19–36. Springer, 2004.
- [119] David Poole. Probabilistic horn abduction and bayesian networks. Artificial intelligence, 64(1):81–129, 1993.
- [120] Peter Haddawy. Generating bayesian networks from probability logic knowledge bases. In Proceedings of the Tenth international conference on Uncertainty in artificial intelligence, pages 262–269. Morgan Kaufmann Publishers Inc., 1994.
- [121] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In IN PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING (ICLP’95). Citeseer, 1995.
- [122] Stephen Muggleton et al. Stochastic logic programs. Advances in inductive logic programming, 32:254–264, 1996.
- [123] Liem Ngo and Peter Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. Theoretical Computer Science, 171(1):147–177, 1997.
- [124] L Getoor and D Jensen. Working notes of the ijcai-2003 workshop on learning statistical models from relational data. SRL-03, 2003.
- [125] Luc De Raedt and Kristian Kersting. Probabilistic logic learning. ACM SIGKDD Explorations Newsletter, 5(1):31–48, 2003.
- [126] Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In IJCAI, volume 97, pages 1330–1339. Citeseer, 1997.
- [127] Andreas Eisele. Towards probabilistic extensions of constraint-based grammars. Computational Aspects of constraint-based linguistic Description II, pages 3–21, 1994.
- [128] James Cussens. Loglinear models for first-order probabilistic reasoning. In Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence, pages 126–133. Morgan Kaufmann Publishers Inc., 1999.
- [129] Kristian Kersting and Luc De Raedt. Towards combining inductive logic programming with bayesian networks. In Inductive Logic Programming, pages 118–131. Springer, 2001.
- [130] Kristian Kersting and Luc De Raedt. Adaptive bayesian logic programs. In Inductive Logic Programming, pages 104–117. Springer, 2001.
- [131] Kristian Kersting, Tapani Raiko, Stefan Kramer, and Luc De Raedt. Towards discovering structural signatures of protein folds based on logical hidden markov models. In Pacific symposium on biocomputing, volume 8, pages 192–203, 2003.

- [132] Kave Eshghi. Abductive planning with event calculus. In ICLP/SLP, pages 562–579, 1988.
- [133] Murray Shanahan. An abductive event calculus planner. The Journal of Logic Programming, 44(1):207–240, 2000.
- [134] Steve Moyle. Using theory completion to learn a robot navigation control program. In Inductive Logic Programming, pages 182–197. Springer, 2003.
- [135] Ramon P Otero. Induction of the indirect effects of actions by monotonic methods. In Inductive Logic Programming, pages 279–294. Springer, 2005.
- [136] Katsumi Inoue, Hideyuki Bando, and Hidetomo Nabeshima. Inducing causal laws by regular inference. In Inductive Logic Programming, pages 154–171. Springer, 2005.
- [137] Nam Tran and Chitta Baral. Hypothesizing about signaling networks. Journal of Applied Logic, 7(3):253–274, 2009.
- [138] Christophe Rodrigues, Pierre Gérard, Céline Rouveirol, and Henry Soldano. Active learning of relational action models. In Inductive Logic Programming, pages 302–316. Springer, 2012.
- [139] Domenico Corapi, Daniel Sykes, Katsumi Inoue, and Alessandra Russo. Probabilistic rule learning in nonmonotonic domains. In Computational Logic in Multi-Agent Systems, pages 243–258. Springer, 2011.
- [140] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. The International Journal of Robotics Research, page 0278364913495721, 2013.
- [141] David Martínez, Guillem Alenyà, and Carme Torras. Relational reinforcement learning with guided demonstrations. Artificial Intelligence, 2015.
- [142] Daniel H Grollman and Odest Chadwicke Jenkins. Dogged learning for robots. In Robotics and Automation, 2007 IEEE International Conference on, pages 2483–2488. IEEE, 2007.
- [143] Sonia Chernova and Manuela Veloso. Interactive policy learning through confidence-based autonomy. Journal of Artificial Intelligence Research, 34(1):1, 2009.
- [144] David Martinez, Guillem Alenya, Pablo Jimenez, Carme Torras, Jurgen Rossmann, Nils Wantia, Eren Erdal Aksoy, Simon Haller, and Justus Piater. Active learning of manipulation sequences. In Robotics and Automation (ICRA), 2014 IEEE International Conference on, pages 5671–5678. IEEE, 2014.
- [145] David Martinez, Guillem Alenya, and Carme Torras. V-min: Efficient reinforcement learning through demonstrations and relaxed reward demands. 2015.

- [146] Chiaki Sakama. Nonmonotomic inductive logic programming. In Logic Programming and Nonmonotonic Reasoning, pages 62–80. Springer, 2001.
- [147] Chiaki Sakama. Induction from answer sets in nonmonotonic logic programs. ACM Transactions on Computational Logic (TOCL), 6(2):203–231, 2005.
- [148] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In ICLP/SLP, volume 88, pages 1070–1080, 1988.
- [149] Chiaki Sakama and Katsumi Inoue. Abduction, unpredictability and garden of eden. Logic Journal of IGPL, page jzt015, 2013.
- [150] AS d’Avila Garcez, Krysia Broda, and Dov M Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. Artificial Intelligence, 125(1):155–207, 2001.
- [151] Jens Lehmann, Sebastian Bader, and Pascal Hitzler. Extracting reduced logic programs from artificial neural networks. Applied intelligence, 32(3):249–266, 2010.
- [152] Artur S Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. Applied Intelligence, 11(1):59–77, 1999.
- [153] Shoudan Liang, Stefanie Fuhrman, and Roland Somogyi. Reveal, a general reverse engineering algorithm for inference of genetic network architectures. 1998.
- [154] Tatsuya Akutsu, Satoru Kuhara, Osamu Maruyama, and Satoru Miyano. Identification of genetic networks by strategic gene disruptions and gene overexpressions under a boolean model. Theoretical Computer Science, 298(1):235–251, 2003.
- [155] Ranadip Pal, Ivan Ivanov, Aniruddha Datta, Michael L Bittner, and Edward R Dougherty. Generating boolean networks with a prescribed attractor structure. Bioinformatics, 21(21):4021–4025, 2005.
- [156] Hannes Klarner, Alexander Bockmayr, and Heike Siebert. Computing symbolic steady states of boolean networks. In Cellular Automata, pages 561–570. Springer, 2014.
- [157] Heike Siebert. Analysis of discrete bioregulatory networks using symbolic steady states. Bulletin of mathematical biology, 73(4):873–898, 2011.
- [158] Tatsuya Akutsu, Takeyuki Tamura, and Katsuhisa Horimoto. Completing networks using observed data. In Algorithmic Learning Theory, pages 126–140. Springer, 2009.
- [159] Alireza Tamaddoni-Nezhad, Raphael Chaleil, Antonis Kakas, and Stephen Muggleton. Application of abductive ilp to learning metabolic network inhibition from temporal data. Machine Learning, 64(1-3):209–230, 2006.

- [160] Katsumi Inoue, Andrei Doncescu, and Hidetomo Nabeshima. Completing causal networks by meta-level abduction. Machine learning, 91(2):239–277, 2013.
- [161] Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. Journal of the ACM (JACM), 23(4):733–742, 1976.
- [162] Michael J Kearns and Umesh Virkumar Vazirani. An introduction to computational learning theory. MIT press, 1994.
- [163] Luc De Raedt. Logical settings for concept-learning. Artificial Intelligence, 95(1):187–201, 1997.
- [164] Hendrik Blockeel, Luc De Raedt, Nico Jacobs, and Bart Demoen. Scaling up inductive logic programming by learning from interpretations. Data Mining and Knowledge Discovery, 3(1):59–93, 1999.
- [165] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. Artificial intelligence, 101(1):285–297, 1998.
- [166] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. Machine learning, 43(1-2):7–52, 2001.
- [167] Howard A Blair, Jagan Chidella, Fred Dushin, Audrey Ferry, and Polar Humenn. A continuum of discrete systems. Annals of Mathematics and Artificial Intelligence, 21(2-4):153–186, 1997.
- [168] Andrew Adamatzky. Identification of cellular automata. In Encyclopedia of Complexity and Systems Science, pages 4739–4751. Springer, 2009.
- [169] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In 21st Network and Distributed System Security Symposium, 2014.
- [170] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In Proceedings of the 16th ACM conference on Computer and communications security, pages 621–634. ACM, 2009.
- [171] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In Security and Privacy, 2009 30th IEEE Symposium on, pages 110–125. IEEE, 2009.
- [172] Alexandre Rocca, Nicolas Mobilia, Éric Fanchon, Tony Ribeiro, Laurent Trilling, and Katsumi Inoue. Asp for construction and validation of regulatory biological networks. Logical Modeling of Biological Systems, pages 167–206.