A BDD-Based Algorithm for Learning from Interpretation Transition

Tony Ribeiro[1], Katsumi Inoue[1,2], and Chiaki Sakama[3]

[1] The Graduate University for Advanced Studies (Sokendai),
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
tony_ribeiro@nii.ac.jp,
[2] National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan,
inoue@nii.ac.jp,
[3] Department of Computer and Communication Sciences,
Sakaedani, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

Abstract. In recent years, there has been an extensive interest in learning the dynamics of systems. For this purpose, a previous work proposed an algorithm that learns a logic program from interpretation transitions. However, both the run time and the memory space of this algorithm alike are exponential. In this paper, we propose a new version of this algorithm utilizing an efficient data structure based on Zero-suppressed Binary Decision Diagrams. We show empirically that using this representation we can perform the same learning task faster and using less memory space.

1 Introduction

In recent years, there has been a notable interest in the field of Inductive Logic Programming (ILP) to learn from system state transitions as part of a wider interest in learning the dynamics of systems [1, 2]. 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 to design more efficient drugs. In some previous works, state transition systems are represented with logic programs [3, 4], 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*) [5, 6].

Which such a bakground, Inoue *et al.* [2] have recently proposed a framework to learn logic programs from traces of interpretation transitions (LFIT). The learning setting of this framework is as follows. We are given a set of pairs of Herbrand interpretations (I, J) as positive examples such that $J = T_P(I)$, and

the goal is to induce a *normal logic program* (NLP) P that realizes the given transition relations. In [2], the authors showed one of the possible usages of LFIT: **LF1T**, *learning from 1-step transitions*. In that paper, an algorithm is proposed to iteratively learn an NLP that realizes the dynamics of the system by considering step transitions one by one. The iterative character of LF1T has applications in bioinformatics, cellular automata, multi-agent systems and robotics. We can easily imagine an agent or a robot that learns the dynamics of its environment from its observations, learning the consequences of its actions 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.

In this paper, we propose a new version of the LF1T algorithm based on Binary Decision Diagrams (BDDs) [7, 8]. A BDD is a canonical representation of a Boolean formula which has been successfully used in many research fields such as Boolean satisfiability solvers [9], data mining [10], ILP [11] and abduction [12, 13]. ProbLog [11] is a probabilistic logic programming language that computes probabilities via BDDs. A ProbLog program computes the probability of a query atom by applying sum-product computation to a BDD, but allows definite clauses only. For abduction in propositional theories, Simon and del Val [12] propose a consequence-finding procedure implemented on Zero-suppressed BDDs. Inoue *et al.* [13] run the EM algorithm over BDDs to evaluate abductive hypotheses.

The main concern of our LF1T algorithm is the size of NLPs learned. For the sake of memory usage and reasoning time, a small NLP could be preferred in multi-agent and robotics applications. In bioinformatics, 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. In previous algorithms, LF1T uses resolution techniques to generalize rules and reduces 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) [14, 15]. 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 LF1T 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 a Zero-suppressed Binary Decision Diagram (ZDD) [16].

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 LF1T. One weak point of the previous LF1T 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 LF1T learning time to the NLP size. Study of the

computational complexity of our new method shows that it remains equivalent to the previous version of LF1T in the worst case. Using examples from the biological literature we show through experimental results that our new algorithm still outperforms the two previous versions of LF1T in practice.

The rest of this paper is organized as follows. Section 2 reviews LF1T together with two previous versions of its algorithms. Section 3 describes the new LF1T algorithm based on BDDs and discusses its computational complexity. Section 4 shows experimental results of the new algorithm compared to the two previous versions of LF1T on learning Boolean networks.

2 Learning from 1-Step Transitions

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

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

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

An (*Herbrand*) *interpretation* I is a subset of $\mathcal{B}$. For a logic program P and an Herbrand interpretation I, the *immediate consequence operator* (or T_P *operator*) [6] is the mapping $T_P : 2^{\mathcal{B}} \to 2^{\mathcal{B}}$:

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

LFIT is an *anytime algorithm* that takes a set of state transitions E as input. From these transitions the algorithm learns a logic program P that represents the dynamics of E. To perform this learning process we can iteratively consider one-step transitions. In LF1T, the Herbrand base $\mathcal{B}$ is assumed to be finite. In the input E, a state transition is represented by a pair of Herbrand interpretations. The output of LF1T is an NLP that realizes all state transitions.

To construct an NLP for LF1T we can use a bottom-up method. A bottom-up method generates hypotheses by *generalization* from the most specific clauses or examples until every positive example is covered. For two rules R_1, R_2 with the same head, R_1 *subsumes* R_2 if there is a substitution θ such that $b^+(R_1)\theta \subseteq b^+(R_2)$ and $b^-(R_1)\theta \subseteq b^-(R_2)$.

The pseudo-code of LF1T is given in algorithm 1. The LF1T algorithm can be used with or without an initial NLP P_0. Given only the examples E, LF1T is initially called by $\mathbf{LF1T}(E, \emptyset)$. If an initial NLP P_0 is given, $\mathbf{LF1T}(E, P_0)$ is called. LF1T firstly constructs the most specific rule R_A^I for each positive literal A appearing in $J = T_P(I)$ for each $(I, J) \in E$. It is important here that *we*

Algorithm 1 LF1T(E,P)

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

P_{old}: NLP
$P_{old} \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
    Pick $(I, J) \in E$; $E := E \setminus \{(I, J)\}$
    for each $A \in J$ **do**
        $R_A^I := A \leftarrow \bigwedge_{B_i \in I} B_i \wedge \bigwedge_{C_j \in (\mathcal{B} \setminus I)} \neg C_j$
        AddRule(R_A^I, P, P_{old})
end while
return P

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 two generalization methods considered in [2] are based on *resolution*. In [2], naïve and ground resolutions are defined between two ground rules as follows. Let R_1, R_2 be two ground 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

$$res(R_1, R_2) = \left(h(R_1) \leftarrow \bigwedge_{L_i \in b(R_1) \setminus \{l\}} L_i \right). \tag{3}$$

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*.

Both naïve resolution and ground resolution can be used as generalization methods of ground rules. For two ground rules R_1 and R_2, the naïve resolution $res(R_1, R_2)$ subsumes both R_1 and R_2, but the non-naïve ground resolution subsumes R_1 only. Ground and naïve resolutions can be used to learn a ground NLP. For example, suppose the three rules: $R_1 = (p \leftarrow q \wedge r)$, $R_2 = (p \leftarrow \neg q \wedge r)$, $R_3 = (p \leftarrow \neg q)$, and their resolvent: $res(R_1, R_2) = res(R_1, R_3) = (p \leftarrow r)$. R_1 and R_2 are complementary with respect to q. Both R_1 and R_2 can be generalized by the naïve resolution of them because $res(R_1, R_2)$ subsumes both R_1 and R_2. On the other hand, the ground resolution $res(R_1, R_3)$ of R_1 and R_3 is equivalent to $res(R_1, R_2)$. However, $res(R_1, R_3)$ subsumes R_1 but does not subsume R_3.

2.1 Generalization by Naïve Resolution

In the first implementation of **LF1T** of [2], naïve resolution is used as a least generalization method. This method is particularly intuitive from the ILP viewpoint, since each generalization is performed based on a least generalization

operator. In [2], it is shown that for two complementary ground rules R_1 and R_2, the naïve resolution of R_1 and R_2 is the least generalization [17] of them, that is, $lg(R_1, R_2) = res(R_1, R_2)$.

When naïve resolution is used, LF1T needs an auxiliary set P_{old} of rules to globally store subsumed rules, which increases monotonically. P_{old} is set to be $\emptyset$ at first. When a generated rule R is newly added, **AddRule**(R, P, P_{old}) tries 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 a rule R', then R is just added to P; otherwise, add R and R' to P_{old} and then add $res(R, R')$ to P by **AddRule**$(res(R, R'), P, P_{old})$.

2.2 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 the second implementation of LF1T of [2], ground resolution is used as an alternative generalization method in **AddRule**. This replacement of resolution leads to a lot of computational gains, since the use of P_{old} is not necessary any more: all generalized rules obtained from $P \cup P_{old}$ by naïve resolution can be obtained using ground resolution on P. In this case, Algorithm 1 is simplified by deleting Lines 3 and 4 and by replacing Line 9 with **AddRule**(R^I_A, P). Here **AddRule** adds and simplifies rules using ground resolution.

By Theorem 3 of [2], using the 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 $O(4^n)$. Given the set E of complete state transitions, which has the size $O(2^n)$, the complexity of **LF1T**$(E, \emptyset)$ with ground resolution is bounded by $O(|E|^2)$. On the other hand, the worst-case complexity of learning with naïve resolution is $O(n^2 \cdot |E|^{4.5})$.

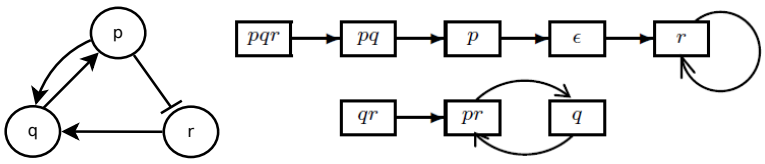

Fig. 1. A Boolean Network N_1(left) and its state transition diagram (right)

Example 1. Consider the state transition in Fig. 1. By giving the state transitions step-by-step and using ground resolution the NLP $\{\#13, \#16, \#19\}$ is obtained in Table 1, where $\#n$ is the rule ID.

Table 1. Execution of **LF1T** with ground resolution on step transitions of figure 1 where $pqr \rightarrow pq$ represents the state transition $(\{p, q, r\}, \{p, q\})$.

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

3 BDD Algorithms for LF1T

Now we present a new LF1T algorithm based on an efficient data structured inspired from OBDD and Zero-suppressed BDD. The novelty of our 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 2 show the evolution of the BDD that represents rules of p in Example 1: 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 2, 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 OBBD, 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 2, 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.

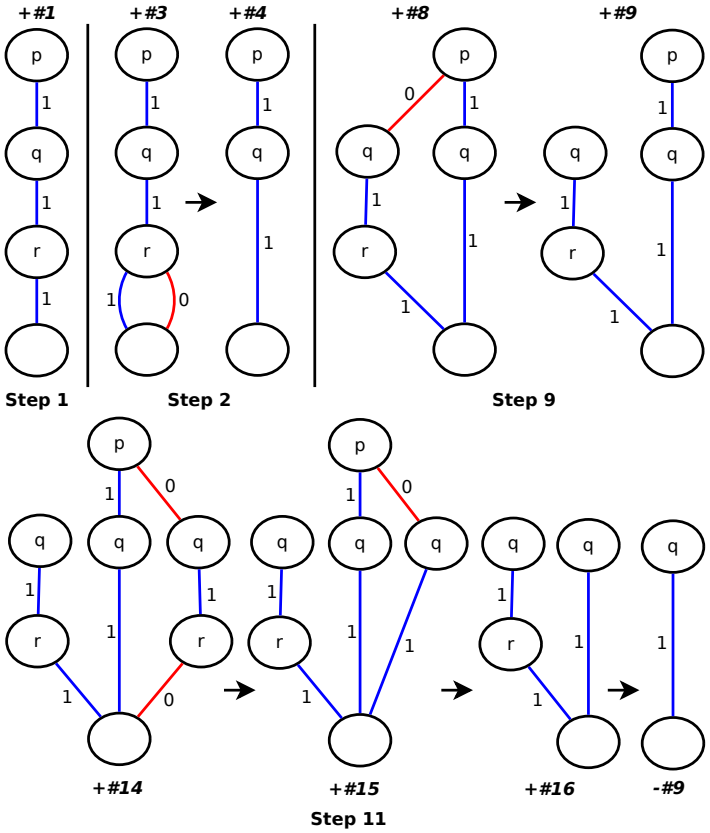

Fig. 2. Evolution of the BDD of p in Example 1, edge labelled by 0 represents negation, nodes without parent are roots and the empty node is the leaf. Last schema of each step represents the real state of the BDD; intermediate ones illustrate update operations. Step 1: from (pqr, pq) we learn $p \leftarrow p \wedge q \wedge r$. 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 subsumtion to finish with $p \leftarrow q$.

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 2, the last BDD of each learning step respects this notion of minimality. Algorithm 2 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.

Algorithm 2 addRule(R,B)

```
1: INPUT: a rule R and a BDD B
2: g: a set of rules
   // 1) Check if R is subsumed
3: for each root node r of B do
4:     if r.subsumes(R, 0) then return
   // 2) Generalizes R
5: for each root node r of B do
6:     if r.generalizes(R, 0) then restart the for loop
   // 3) Remove rules subsumed by R
7: l := the leaf node of B
8: l.clear(R, |R|, true)
   // 4) Insert R into the BDD
9: insert(R, B)
   // 5.1) Check generalization by R
10: g ← ∅
11: for each root node r of B do
12:     r.generalizations(R, 1, g)
   // 5.2) Add the generalizations generated by R
13: for each rules R_g of g do
14:     addRule(R_g)
```

The details of each step is explained as follows.

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 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.

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 subsumes and is not subsumed by any rules of B and cannot be generalized by a rule of B using ground 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's 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 LF1T.

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 1. *Let n be the size of the Herbrand base $|\mathcal{B}|$. Using our dedicated BDD structure the memory complexity as well as the computational complexity of LF1T remain in the same order as the previous algorithm based on ground resolution: , i.e., $O(2^n)$ and $O(4^n)$, respectively. The proof is given as appendix.*

4 Experiments

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 [18], 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` [19]. Finally, we use this set 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 transition, that is, $T_{P_{org}}$ and $T_{P_{LFIT}}$ are identical functions.

Table 2. Memory use and learning time of **LF1T** for Boolean networks up to 15 nodes with the alphabetical variable ordering

Name	# nodes	# rules	Naïve	Ground	BDD
Arabidopsis thalania	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 2 shows the memory space and time of a single LF1T run in learning a Boolean network for each problem in [18] 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 show more precise experimental results on the BDD version of LF1T. This table shows the minimimum, 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.

Table 3. Experimental results of 1000 runs of **LF1T** with random variable orderings

Name	min/max # rules	Average # rules	time	std deviation rules/time
Arabidopsis thalania	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

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 should also affect the previous version of LF1T, but since such study has not been done on the previous version we cannot compare our results on that point.

5 Conclusion

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.

Just a few remarks on learning non-ground NLPs; LF1T first learns ground rules then we apply well-known generalization techniques like *anti-instantiation* and least generalization. Extension of the BDD structure in this paper to the first-order case like [20] remains as a future work. Another possible outlook is an extension of LF1T algorithm to learn the dynamics of asynchronous systems.

References

1. Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K., Srinivasan, A.: Ilp turns 20. Machine learning **86**(1) (2012) 3–23
2. Inoue, K., Ribeiro, T., Sakama, C.: Learning from interpretation transition. Machine Learning (2013) doi: 10.1007/s10994–013–5353–8
3. Inoue, K.: Logic programming for boolean networks. In: Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two, AAAI Press (2011) 924–930
4. Inoue, K., Sakama, C.: Oscillating behavior of logic programs. In: Correct Reasoning. Springer (2012) 345–362
5. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. Journal of the ACM (JACM) **23**(4) (1976) 733–742
6. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. Foundations of deductive databases and logic programming (1988) 89
7. Akers, S.B.: Binary decision diagrams. Computers, IEEE Transactions on **100**(6) (1978) 509–516
8. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on **100**(8) (1986) 677–691
9. Aloul, F.A., Mneimneh, M.N., Sakallah, K.A.: Zbdd-based backtrack search sat solver. In: Proc. Intl Workshop on Logic Synthesis, Lake Tahoe, California. (2002)
10. Minato, S., Arimura, H.: Frequent closed item set mining based on zero-suppressed bdds. Information and Media Technologies **2**(1) (2007) 309–316
11. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: A probabilistic prolog and its application in link discovery. In: Proceedings of the 20th international joint conference on Artifical intelligence. (2007) 2468–2473
12. Simon, L., Del Val, A.: Efficient consequence finding. In: International Joint Conference on Artificial Intelligence. Volume 17., LAWRENCE ERLBAUM ASSOCIATES LTD (2001) 359–370
13. Inoue, K., Sato, T., Ishihata, M., Kameya, Y., Nabeshima, H.: Evaluating abductive hypotheses using an em algorithm on bdds. In: Proceedings of the 21st international jont conference on Artifical intelligence, Morgan Kaufmann Publishers Inc. (2009) 810–815
14. Bryant, R.E., Meinel, C.: Ordered binary decision diagrams. Springer (2002)
15. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR) **24**(3) (1992) 293–318
16. Minato, S.: Zero-suppressed bdds for set manipulation in combinatorial problems. In: 30th Conference on Design Automation, IEEE (1993) 272–277
17. Plotkin, G.D.: A note on inductive generalization. Machine intelligence **5**(1) (1970) 153–163
18. Dubrova, E., Teslenko, M.: A sat-based algorithm for finding attractors in synchronous boolean networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) **8**(5) (2011) 1393–1399
19. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers (2012)
20. Groote, J.F., Tveretina, O.: Binary decision diagrams for first-order predicate logic. The Journal of Logic and Algebraic Programming **57**(1) (2003) 1–22
21. Liaw, H.T., Lin, C.S.: On the obdd-representation of general boolean functions. IEEE Trans. Computers **41**(6) (1992) 661–664

A Appendixes

A.1 Proof of Theorem 1

Let n be the size of the Herbrand base $|\mathcal{B}|$. Using our dedicated BDD structure the memory complexity as well as the computational complexity of LF1T remain in the same order as the previous algorithm based on ground resolution: , i.e., $O(2^n)$ and $O(4^n)$, respectively. The proof is given as appendix.

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 a NLP $|P|$ learned by LF1T 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 [21]. Because BDD 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: $O(2^n)$.

Regarding learning, each operation has its own complexity. Let k be the place of a literal in the variable ordering so that for the root 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 subsumed 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 $O(4^n)$.

A.2 Complete Pseudo Code of LF1T Using BDD

In this section we give the complete pseudo code of our new LF1T algorithm. Following algorithms correspond to the operations used in the Algorithm 2.

Algorithm 3 subsumes(R, n)

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

3: literal_N: node literal
4: children_true: list of child nodes where the node literal is true
5: children_false: list of child nodes where the node literal is false
6: head: the head literal of R
   // 1) Leaf node
7: if is_leaf() AND variable = head then
8:     return true
   // 2) End of the rule
9: if n > |R| then
10:    return false
11: literal_R ← n^th literal of R
   // 3) BDD rules are more generals
12: if literal_R > literal_N then
13:    return subsumes(R, n + 1)
14: literal_R ← n^th literal of R
   // 4) The rule is more general
15: if literal_R < literal_N then
16:    return false
   // 5) Same literal
17: if literal_R is positive then
18:    children ← children_true
19: else
20:    children ← children_false
21: for each child node c of children do
22:    if c.subsumes(R, n + 1) then
23:        return true
24: return false
```

Algorithm 4 generalizes(R, n)

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

3: literal_N: node literal
4: children_true: list of child nodes where the node literal is true
5: children_false: list of child nodes where the node literal is false
   // 1) The rule is more general than all rules of the node
6: if n > |R| then return false
   // 2) Leaf node
7: if is_leaf() then return false
   // 3) Check generalization on the current node
8: literal_R ← n^th literal of R
   // 3.1) The node is more general than the rule
9: while literal_N > literal_R do
10:     if subsumes(R, n) then
11:         R ← R \ literal_R // 3.1.1) The node subsumes the complementary rule
12:         return true
13:     n ← n + 1
          // 3.1.2) No more literal to generalize
14:     if n > |R| then return false
15: end while
   // 3.2) The rule is more general
16: if literal_N < literal_R then return false
   // 3.3) The sub-bdd possibly contains the complementary
17: same ← children_true
18: oposite ← children_false
19: if literal_R is positive then
20:     same ← children_false
21:     oposite ← children_true
   // 3.3.1) Search for complementary rules
22: for each child node c of oposite do
23:     if c.subsumes(R, n + 1) then // Complementary rules is subsumed
24:         R ← R \ literal_R
25:         return true
   // 4) Search for generalizations on next literal
26: for each child node c of same do
27:     if c.generalizes(R, n + 1) then
28:         return true
29: return false
```

Algorithm 5 clear(R, n,can_cut)

```
INPUT: R a rule, n an integer and can_cut a boolean
OUTPUT: a Boolean value

literal_R: the n^th literal of R
unlink ← false
// 1) Choice node
if #child > 1 then
    can_cut ← false
// 2) Check parents
for each parent node p do
    literal_p ← the literal of p
// 2.1) Parent is more general
    if literal_p < literal_R then
        if n = 1 AND is_leaf() then
            CONTINUE // 2.1.1) Not subsumed
        if !p.clear(R, n, can_cut) then
            CONTINUE
        2.1.2) Subsumed
        if can_cut then
            remove the link with p
            delete p if it do not has child
            unlink ← true
            CONTINUE
        return true
    2.2) Rule is more general
    if literal_p > literal_R then
        if !p.clear(R, n, can_cut) then
            delete p if it do not has any parent
            CONTINUE // 2.2.1) Not subsumed
        // 2.2.2) Subsumed
        if can_cut then
            remove the link with p
            delete p if it do not has any child
            unlink ← true
            CONTINUE
        return true
    // 2.3) Same literal
    if n > 0 AND !p.clear(R, n − 1, can_cut) then
        delete p if it do not has any parent
        CONTINUE
    // 2.3.2) Subsumed
    if can_cut then
        remove the link with p
        delete p if it do not has any child
        unlink ← true
        CONTINUE
    return true
return false
```

Algorithm 6 insert(R, BDD)

INPUT: a rule R and a BDD

$roots$: the set of root nodes of BDD
$literal$: first literal of R
$begin, end$: BDD nodes
$n \leftarrow 0$
$push \leftarrow false$
// 1) Bottom-up search for common part
$end \leftarrow$ the last ancestor node reached following R from the corresponding leaf node
// 2) Fact rule
if $|R| = 0$ **then**
    $roots \leftarrow roots \cup leaf$
$begin \leftarrow NULL$
// 2.1) Search common literal within the roots
if a node $r \in roots$ correspond to $literal$ **then**
    $begin \leftarrow r$
// 2.2) New root
if $begin = NULL$ **then**
    $begin \leftarrow$ a new node corresponding to $literal$
    $roots \leftarrow roots \cup begin$
    $push \leftarrow true$
$current$: bdd node **pointer**
make $current$ points on $begin$
// 3) Insertion of the rest of the body
while $n \leq |R|$ **do**
    $n \leftarrow n + 1$
    // 3.1) Link node reached
    if $n > |R|$ OR the n^{th} literal of R is the one of end **then**
        connect $current$ to end according to the polarity of $literal$
        return
    $literal \leftarrow n^{th}$ literal of R
// 3.2) Push chain
    if push **then**
        create a new node for $literal$
        connect the node to $current$ according to the polarity of $literal$
        make $current$ points on the new node
        CONTINUE
// 3.3) Continue to follow the rule
    $next \leftarrow NULL$
    for each child nodes c of $current$ according to previous $literal$ polarity **do**
        if c has only one parent node AND correspond to $literal$ **then**
            $next \leftarrow c$
            BREAK
// 3.4) // No more common literal
    if $next = NULL$ **then**
        $push = true$
        $n \leftarrow n - 1$
        CONTINUE
// 3.4) // Continue to follow the BDD
    Make $current$ point on $next$
end while
Connect end to $begin$ according to the polarity of $literal$

Algorithm 7 generalizations(R, n,G)

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

$literal_N$: node literal
$G', rules$: set of rules
// 1) End of the rule
if $n > |R|$ **then return**
$literal_R \leftarrow n^{th}$ literal of R
// 2) Node is more general
if $literal_N > literal_R$ **then return**
// 3) Generalizations are possible on all children
if $literal_N < literal_R$ **then**
    for each child node c **do**
        $rules \leftarrow$ all rules subsumed by R in c
        $G \leftarrow G \cup rules$
2.2) Retrieve deeper generalizations
    for each child node c **do**
        $G' \leftarrow \emptyset$
        $c.generalizations(R, n+1, G')$
        $literal \leftarrow literal_N$
        if the link with c is a negation **then**
            $literal \leftarrow \neg literal_N$
        for each rule r of G' **do**
            $G \leftarrow G \cup \{(h(r) \leftarrow literal \wedge \bigwedge_{l \in b(r)} l)\}$
    return
// 3) Same literal
for each child node c **do**
    // 3.1) Search complementary rules
    if the link with c has the same polarity as $literal_R$ **then**
        $rules \leftarrow$ all rules subsumed by R in c
        $G \leftarrow G \cup rules$
    else
        // 3.2) Check deeper generalizations
        $literal \leftarrow literal_N$
        if the link with c is a negation **then**
            $literal \leftarrow \neg literal_N$
        $G' \leftarrow \emptyset$
        $c.generalizations(R, n+1, G')$
        for each rule r of G' **do**
            $G \leftarrow G \cup \{(h(r) \leftarrow literal \wedge \bigwedge_{l \in b(r)} l)\}$