Differentiable Inductive Logic Programming for Structured Examples

Hikaru Shindo[1], **Masaaki Nishino**[2], **Akihiro Yamamoto**[1]
[1] Kyoto University
[2] NTT Communication Science Laboratories, NTT Corporation
shindo@iip.ist.i.kyoto-u.ac.jp, masaaki.nishino.uh@hco.ntt.co.jp, akihiro@i.kyoto-u.ac.jp

Abstract

The differentiable implementation of logic yields a seamless combination of symbolic reasoning and deep neural networks. Recent research, which has developed a differentiable framework to learn logic programs from examples, can even acquire reasonable solutions from noisy datasets. However, this framework severely limits expressions for solutions, e.g., no function symbols are allowed, and the shapes of clauses are fixed. As a result, the framework cannot deal with structured examples. Therefore we propose a new framework to learn logic programs from noisy and structured examples, including the following contributions. First, we propose an adaptive clause search method by looking through structured space, which is defined by the generality of the clauses, to yield an efficient search space for differentiable solvers. Second, we propose for ground atoms an enumeration algorithm, which determines a necessary and sufficient set of ground atoms to perform differentiable inference functions. Finally, we propose a new method to compose logic programs softly, enabling the system to deal with complex programs consisting of several clauses. Our experiments show that our new framework can learn logic programs from noisy and structured examples, such as sequences or trees. Our framework can be scaled to deal with complex programs that consist of several clauses with function symbols.

Introduction

Integrating symbolic reasoning and numerical computation is increasingly becoming a vital factor in artificial intelligence and its applications (De Raedt et al. 2020). Due to the success of deep neural networks (DNNs), one of the main integrated techniques is to combine DNNs with logical reasoning, which is called neuro-symbolic computation (d'Avila Garcez et al. 2019). The main goal is to establish a unified framework that can make flexible approximations using DNNs and perform tractable and multi-hop reasoning using first-order logic.

Although many approaches have been developed for the integration of logic and DNNs (Rocktäschel and Riedel 2017; Yang, Yang, and Cohen 2017; Šourek et al. 2018; Manhaeve et al. 2018; Si et al. 2019; Cohen, Yang, and Mazaitis 2020; Riegel et al. 2020; Marra et al. 2020), most existing approaches involve the learning of continuous parameters, not discrete structures. Structure learning (Kok and Domingos 2005), in which logical expressions are obtained explicitly, presents a challenge to neuro-symbolic approaches (De Raedt et al. 2020).

Evans and Grefenstette proposed (Evans and Grefenstette 2018) Differentiable Inductive Logic Programming (∂ILP), which is a framework for learning logic programs from given examples in a differentiable manner. Inductive Logic Programming (ILP) (Muggleton 1991) is a sound formalization for finding theories from given examples using first-order logic as its language (Nienhuys-Cheng et al. 1997). The ∂ILP framework formulates ILP problems as numerical optimization problems that can be solved by gradient descent. Its differentiability establishes a seamless combination of ILP and neural networks to deal with subsymbolic and noisy data.

However, previous work has put severe limitations on expressions for solutions. For instance, no function symbols are allowed, the arity of predicates must be less than 2, the number of atoms in the clause body must not exceed 2, and every program must be comprised of pairs of rules for each predicate. Thus it is unsuitable for complex structured data, such as sequences or trees, or complex programs that are comprised of several clauses for a predicate. One main characteristic of first-order logic is the expressibility and learnability for structured data with function symbols (Lloyd 2003; Dantsin et al. 2001; Fredouille et al. 2007). We face $three$ main challenges to deal with complex programs and structured data: (i) the number of clauses to be considered increases, (ii) an infinite number of ground atoms can be generated with function symbols, and (iii) the memory and computation costs increase quadratically with respect to the size of the search space. We address these issues by proposing a new differentiable approach to learning logic programs by combining adaptive symbolic search methods and continuous optimization methods and make the following contributions for each problem:

Clause Search with Refinement We propose an efficient clause search method for a differentiable ILP solver. We generate clauses by beam searching and leveraging the generality of clauses and the given examples. We start from general (strong) clauses and incrementally specify (weaken) the clauses. We only take clauses that contribute to accu-

rate classification results into the search space. Our approach yields an efficient search space that includes only promising clauses for the differentiable ILP solver.

Adaptive Fact Enumeration We present a fact enumeration algorithm to implement the differentiable inference function. The number of ground atoms defines the size of the tensors used in the differentiable step, thus the set of required ground atoms must be determined. We enumerate the ground atoms using the given examples and the generated clauses by backward-chaining. Our approach yields a small set of ground atoms, and this small set is a key factor to achieve differentiable learning from structured objects.

Soft Program Composition We propose a practical algorithm to learn complex logic programs in a differentiable manner. In past studies, the weights were assigned to each pair of clauses because some information is lost if weights are assigned to each clause, and thus the number of parameters increased quadratically. In our approach, we compose a differentiable inference function by assigning multiple distinct weights to each clause and introducing a function to compute logical *or* softly. Our approach efficiently estimates logic programs in terms of memory and computation costs.

Notation We use bold lowercase letters $\mathbf{v}, \mathbf{w}, \ldots$ for vectors and the functions that return vectors. We use bold capital letters $\mathbf{X}, \ldots$ for tensors. We use calibrate letters $\mathcal{C}, \mathcal{A}, \ldots$ for sets and ordered sets.

Related Work

A pioneering study of inductive inference was done in the early 70s (Plotkin 1971). The Model Inference System (MIS) (Shapiro 1983) has been implemented as an efficient search algorithm for logic programs using the generality of expressions. Inductive Logic Programming (Muggleton 1991) has emerged at the intersection of machine learning and logic programming. The Elementary Formal System (EFS) (Arikawa, Shinohara, and Yamamoto 1992) is a well-established system for strings based on first-order logic.

Dealing with uncertainty in ILP has been a major obstacle. Probabilistic Inductive Logic Programming (De Raedt and Kersting 2004) combines probability theory with ILP. It is also known as Statistical Relational Learning (De Raedt et al. 2016). Another approach to cope with uncertainty is to combine neural methods with differentiable implementations of logic (Rocktäschel and Riedel 2017; Yang, Yang, and Cohen 2017; Evans and Grefenstette 2018; Šourek et al. 2018; Manhaeve et al. 2018; Si et al. 2019; Riegel et al. 2020; Cohen, Yang, and Mazaitis 2020; Marra et al. 2020). Both of these approaches blazed a trail for the integration of logic, probability, and neural methods. However, almost all of these approaches are domain-specific (De Raedt et al. 2020), i.e., the expressions are severely limited. A critical gap exists between these past approaches and logic-based systems for structured data, such as MIS and EFS. Our work fills the gap by incorporating symbolic methods with differentiable approaches.

A propositional approach for ILP is one established approach, which was developed to integrate ILP and SAT solvers or Binary Decision Diagrams (Chikara et al. 2015; Shindo, Nishino, and Yamamoto 2018). The ∂ILP system performed differentiable learning by incorporating continuous relaxation into these approaches. We also follow this approach.

Beam searching with clause refinement was developed for structure learning for probabilistic logic programs (Bellodi and Riguzzi 2015; Nguembang Fadja and Riguzzi 2019). We use this approach because it requires fewer declarative biases than approaches based only on templates.

Inductive Logic Programming Concepts

Basic Concepts *Language* $\mathcal{L}$ is a tuple $(\mathcal{P}, \mathcal{F}, \mathcal{A}, \mathcal{V})$, where $\mathcal{P}$ is a set of predicates, $\mathcal{F}$ is a set of function symbols, $\mathcal{A}$ is a set of constants, and $\mathcal{V}$ is a set of variables. We denote n-ary predicate p by p/n and n-ary function symbol f by f/n. A *term* is a constant, a variable, or an expression $f(t_1, \ldots, t_n)$ where f is a n-ary function symbol and $t_1, \ldots, t_n$ are terms. A function symbol yields a structured expression. An *atom* is a formula $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. A *ground atom* or simply a *fact* is an atom with no variables. A *literal* is an atom or its negation. A *positive literal* is just an atom. A *negative literal* is the negation of an atom. A *clause* is a finite disjunction ($\vee$) of literals. A *definite clause* is a clause with exactly one positive literal. If $A, B_1, \ldots, B_n$ are atoms, then $A \vee \neg B_1 \vee \ldots \vee \neg B_n$ is a definite clause. We write definite clauses in the form of $A \leftarrow B_1 \wedge \ldots \wedge B_n$. Atom A is called the *head*, and set of negative atoms $\{B_1, \ldots, B_n\}$ is called the *body*. We denote special constant *true* as $\top$ and *false* as $\bot$. We denote a set of variables in clause C as $V(C)$. $DV_n(C)$ is a set of all n-combinations of distinct variables in clause C, i.e., $DV_n(C) = \{(x_1, \ldots, x_n) | x_i \in V(C) \wedge x_i \neq x_j (i \neq j)\}$. Substitution $\theta = \{x_1 = t_1, ..., x_n = t_n\}$ is an assignment of term t_i to variable x_i. An application of substitution θ to atom A is written as $A\theta$. A *unifier* for the set of expressions $\{A_1, \ldots, A_n\}$ is a substitution θ such that $A_1\theta = A_2\theta = \ldots = A_n\theta$, written as $\theta = \sigma(\{A_1, \ldots, A_n\})$, where σ is a *unification function.* A unification function returns the (most general) unifier for the expressions if they are unifiable. Decision function $\bar{\sigma}(\{A_1, \ldots, A_n\})$ returns a Boolean value whether or not $A_1, \ldots, A_n$ are unifiable.

Inductive Logic Programming

ILP problem $\mathcal{Q}$ is tuple $(\mathcal{E}^+, \mathcal{E}^-, \mathcal{B}, \mathcal{L})$, where $\mathcal{E}^+$ is a set of positive examples, $\mathcal{E}^-$ is a set of negative examples, $\mathcal{B}$ is background knowledge, and $\mathcal{L}$ is a language. We assume that the examples and the background knowledge are ground atoms. The solution to an ILP problem is a set of definite clauses $\mathcal{H} \subseteq \mathcal{L}$ that satisfies the following conditions:

- $\forall A \in \mathcal{E}^+ \; \mathcal{H} \cup \mathcal{B} \models A$.
- $\forall A \in \mathcal{E}^- \; \mathcal{H} \cup \mathcal{B} \not\models A$.

Typically the search algorithm starts from general clauses. If the current clauses are too general (strong), i.e., they entail too many negative examples, then the solver incrementally specifies (weakens) them. This weakening operation is called a *refinement*, which is one of the essential tools for ILP.

Refinement Operator The refinement operator defines between clauses the complexity that varies from *general* to *specific*. The refinement operator takes a clause and returns weakened clauses. Generally, there are four types of refinement operators: (i) application of function symbols, (ii) substitution of constants, (iii) replacement of variables, and (iv) addition of atoms. For clause $C = A \leftarrow B_1, \ldots, B_m$, each refinement operation for language $\mathcal{L} = (\mathcal{P}, \mathcal{F}, \mathcal{A}, \mathcal{V})$ is as follows:

- For $z \in V(C)$, $f \in \mathcal{F}$, and $x_1, \ldots, x_n \in \mathcal{V} \backslash V(C)$, let $C\{z = f(x_1, \ldots, x_n)\} \in \rho_{\mathcal{L}}^{fun}(C)$, where $x_1, \ldots, x_n$ are pairwise different.
- For $z \in V(C)$ and $a \in \mathcal{A}$, let $C\{z = a\} \in \rho_{\mathcal{L}}^{sub}(C)$.
- For $z, y \in V(C)$ $(z \neq y)$, let $C\{z = y\} \in \rho_{\mathcal{L}}^{rep}(C)$.
- For n-ary predicate $p \in \mathcal{P}$ and $(x_1, \ldots, x_n) \in DV_n(C)$, let $A \leftarrow B_1, \ldots, B_m, p(x_1, \ldots, x_n) \in \rho_{\mathcal{L}}^{add}(C)$.

The refinement operator for language $\mathcal{L}$ is defined:

$$\rho_{\mathcal{L}}(C) = \rho_{\mathcal{L}}^{func}(C) \cup \rho_{\mathcal{L}}^{subs}(C) \cup \rho_{\mathcal{L}}^{rep}(C) \cup \rho_{\mathcal{L}}^{add}(C). \quad (1)$$

Example 1 Let $\mathcal{L} = (\mathcal{P}, \mathcal{F}, \mathcal{A}, \mathcal{V})$, where $\mathcal{P} = \{p/2, q/2\}$, $\mathcal{F} = \{f/1\}$, $\mathcal{A} = \{a, b\}$ and $\mathcal{V} = \{x, y, z\}$. Let $\mathcal{E}^+ = \{p(a,a), p(b,b)\}$, $\mathcal{E}^- = \{p(a,b), p(b,a)\}$, $\mathcal{B} = \{\}$. One of the solutions is $\mathcal{H} = \{p(x,x)\}$.

Example 2 Let $\mathcal{L}$ be the language specified in Example 1. The following is the result of the refinement: $\rho_{\mathcal{L}}(p(x,y)) = \{p(a,y),\ p(x,a),\ p(b,y),\ p(x,b)\ p(x,x),\ p(f(z),y), p(x,f(z)), p(x,y) \leftarrow q(x,y)\ \}$.

Differentiable Inductive Logic Programming

In the ∂ILP framework (Evans and Grefenstette 2018), an ILP problem is formulated as an optimization problem that has the following general form:

$$\min_{\mathcal{W}} L(\mathcal{Q}, \mathcal{C}, \mathcal{W}), \quad (2)$$

where $\mathcal{Q}$ is an ILP problem, $\mathcal{C}$ is a set of clauses specified by templates, $\mathcal{W}$ is a set of weights for clauses, and L is a loss function that returns a penalty when training constraints are violated. We briefly summarize the steps of the process as follows:

Step 1 Set of ground atoms $\mathcal{G}$ is specified by given language $\mathcal{L} \in \mathcal{Q}$.

Step 2 Tensor $\mathbf{X}$ is built from given set of clauses $\mathcal{C}$ and fixed set of ground atoms $\mathcal{G}$. It holds the relationships between clauses $\mathcal{C}$ and ground atoms $\mathcal{G}$. Its dimension is proportional to $|\mathcal{C}|$ and $|\mathcal{G}|$.

Step 3 Given background knowledge $\mathcal{B} \in \mathcal{Q}$ is compiled into vector $\mathbf{v}_0 \in \mathbb{R}^{|\mathcal{G}|}$. Each dimension corresponds to each ground atom $G_j \in \mathcal{G}$, and $\mathbf{v}_0[j]$ represents the valuation of G_j.

Step 4 A computational graph is constructed from $\mathbf{X}$ and $\mathcal{W}$. The weights define probability distributions over clauses $\mathcal{C}$. A probabilistic forward-chaining inference is performed by the forwarding algorithm on the computational graph with input $\mathbf{v}_0$.

Step 5 The loss is minimized with respect to weights $\mathcal{W}$ by gradient descent techniques. After minimization, a human-readable program is extracted by discretizing the weights.

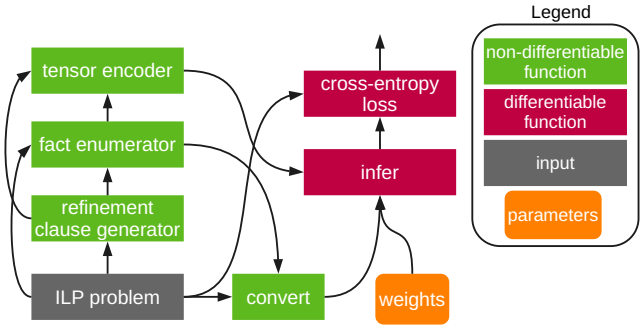

Figure 1: Overview of our model

Method

Although we begin by following the ∂ILP approach, we introduce several new algorithms to deal with structured examples and complex programs with function symbols. An overview of our approach is illustrated in Fig. 1. First, we generate clauses by beam searching with refinement to specify an efficient search space. Second, we enumerate ground atoms by backward-chaining using the set of generated clauses. This enumeration results in efficient inference computation because the number of ground atoms determines the dimensions of tensors for the differentiable steps. Third, we propose a new approach to softly compose complex logic programs. We assign several weights for each clause to define several probability distributions over the clauses and efficiently estimate logic programs that consist of several clauses.

Clause Search with Refinement

We incrementally generate candidates of clauses by refinement and beam searching. Promising clauses for an ILP problem are those that entail many positive examples but few negative examples. Algorithm 1 is our generation algorithm. The inputs are initial clauses $\mathcal{C}_0$, ILP problem $\mathcal{Q}$, the size of the beam in search N_{beam}, and the number of steps of beam searching T_{beam}. We start from the initial clauses and iteratively weaken the top-N_{beam} clauses based on how many positive examples can be entailed by clause combining with background knowledge. The following is the evaluation function for clause R:

$$eval(R, \mathcal{Q}) = |\{E \mid E \in \mathcal{E}^+ \wedge \mathcal{B} \cup \{R\} \models E\}|, \quad (3)$$

where $\mathcal{E}^+$ is a set of positive examples.

The key difference from ∂ILP is that we leverage the given examples to specify the search space for the differentiable solver. In ∂ILP, since the clauses are generated only by templates many meaningless clauses tend to be generated.

Example 3 Let $\mathcal{E}^+ = \{p(a,a),\ p(b,b),\ p(b,c),\ p(c,b)\}$, $\mathcal{B} = \{q(b,c), q(c,b)\}$, $\mathcal{C}_0 = \{p(x,y)\}$, $T_{beam} = 2$, and $N_{beam} = 2$. Fig. 2 illustrates an example of beam searching for this problem. In the 2nd layer, we show examples

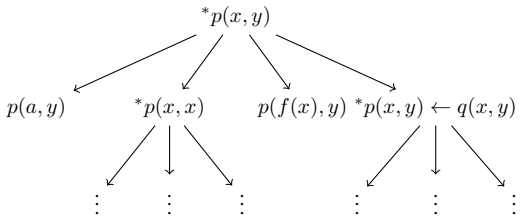

Figure 2: Beam searching for clauses

Algorithm 1 Clause generation by beam searching

Input: $\mathcal{C}_0, \mathcal{Q}, N_{beam}, T_{beam}$
$\mathcal{C}_{to\_open} \leftarrow \mathcal{C}_0$
$\mathcal{C} \leftarrow \emptyset$
$t = 0$
while $t < T_{beam}$ **do**
  $\mathcal{C}_{beam} \leftarrow \emptyset$
  for $C_i \in \mathcal{C}_{to\_open}$ **do**
    $\mathcal{C} = \mathcal{C} \cup \{C_i\}$
    for $R \in \rho_{\mathcal{L}}(C_i)$ **do**
      $score = eval(R, \mathcal{Q})$ //Evaluate each clause
      $\mathcal{C}_{beam} = insert(\mathcal{C}_{beam}, R, score)$ //Insert refined clause in order of scores possibly discarding it
  $\mathcal{C}_{to\_open} = \mathcal{C}_{beam}$ //top-N_{beam} clauses are refined in the next loop
  $t = t + 1$
return $\mathcal{C}$

of generated clauses by refining the initial clause. Each new clause is evaluated and selected to be refined. In this case, clause $p(x,x)$ and $p(x,y) \leftarrow q(x,y)$ is refined in the next step because it entails more positive examples with background knowledge $\mathcal{B}$ than other clauses. Refined clauses are added to set $\mathcal{C}$. By contrast, since clause $p(f(x),y)$ does not entail any positive examples with background knowledge $\mathcal{B}$, it is discarded. Finally, we get set of clauses $\mathcal{C} = \{p(x,y), p(x,x), p(x,y) \leftarrow q(x,y)\}$.

Adaptive Fact Enumeration

We enumerate ground atoms using the given clauses and examples. Algorithm 2 is our enumeration algorithm. The inputs are ILP problem $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-, \mathcal{B}, \mathcal{L})$, set of clauses $\mathcal{C}$, and time-step parameter T that determines the number of forward-chaining steps in the differentiable inference. We start from the given examples, the background knowledge, and special symbols that represent *true* and *false* respectively. We unify the head of each clause and each ground atom. If they are unifiable, then we compute the ground atoms on the body by applying the unifier. Here we assume that the body has fewer variables than the head.

The key difference from ∂ILP is that we utilize the given ILP problem to specify the set of ground atoms. In ∂ILP, the solver considers all the visible ground atoms, which is known as the *Herbrand Base*. However, since an infinite number of ground atoms are yielded by function symbols, it is unsuitable for the case with function symbols.

Example 4 Let $\mathcal{E}^+ = \{e(s^6(0))\}$, $\mathcal{E}^- = \{e(s(0))\}$, $\mathcal{B} = \{e(0)\}$, $\mathcal{C} = \{e(s^2(x)) \leftarrow e(x)\}$, and $T = 2$.

Algorithm 2 Enumeration of ground atoms

Input: $\mathcal{Q}, \mathcal{C}, T$
1: $\mathcal{G} \leftarrow \{\bot, \top\} \cup \mathcal{E}^+ \cup \mathcal{E}^- \cup \mathcal{B}$
2: **for** $i = 0$ to $T - 1$ **do**
3: $\mathcal{S} \leftarrow \emptyset$
4: **for** $A \leftarrow B_1, \ldots, B_n$ in $\mathcal{C}$ **do**
5: **for** $G \in \mathcal{G}$ **do**
6: **if** $\bar{\sigma}(A, G)$ **then**
7: $\theta \leftarrow \sigma(A, G)$
 $\mathcal{S} \leftarrow \mathcal{S} \cup \{B_1\theta, \ldots, B_n\theta\}$
8: $\mathcal{G} \leftarrow \mathcal{G} \cup \mathcal{S}$
9: **return** $\mathcal{G}$

First $\mathcal{G}$ is initialized as $\mathcal{G} = \{\bot, \top, e(s^6(0)), e(s(0)), e(0)\}$. Atom $e(s^6(0))$ and clause head $e(s^2(x))$ are unifiable with $\theta = \{x = s^4(0)\}$. Then body $e(x)\theta = e(s^4(0))$, and this ground atom is added to $\mathcal{G}$. In the next step, atom $e(s^4(0))$ and clause head $e(s^2(x))$ are unifiable with $\theta = \{x = s^2(0)\}$. Hence body $e(x)\theta = e(s^2(0))$ is added to $\mathcal{G}$. Finally, the enumeration algorithm returns $\mathcal{G} = \{\bot, \top, e(0), e(s(0)), e(s^2(0)), e(s^4(0)), e(s^6(0))\}$. Note that ground atoms $e(s^3(0))$ and $e(s^5(0))$ are not required in this case.

Soft Program Composition

Tensor Encoding We build a tensor that holds the relationships between clauses $\mathcal{C}$ and ground atoms $\mathcal{G}$. We assume that $\mathcal{C}$ and $\mathcal{G}$ are an ordered set, i.e., where every element has its own index. Let b be the maximum body length in $\mathcal{C}$. Index tensor $\mathbf{X} \in \mathbb{N}^{|\mathcal{C}|\times|\mathcal{G}|\times b}$ contains the indexes of the ground atoms to compute forward inferences. Intuitively, $\mathbf{X}[i,j] \in \mathbb{N}^b$ contains a set of the indexes of the subgoals to entail the j-th fact using the i-th clause. For clause $C_i = A \leftarrow B_1, \ldots, B_n \in \mathcal{C}$ and set of ground atoms $\mathcal{G}$, we compute tensor $\mathbf{X}$:

$$\mathbf{X}[i,j,k] = \begin{cases} I_{\mathcal{G}}(B_k\theta) \text{ if } \bar{\sigma}(\{A, G_j\}) \wedge k \leq n \\ I_{\mathcal{G}}(\top) \text{ if } \bar{\sigma}(\{A, G_j\}) \wedge k > n \\ I_{\mathcal{G}}(\bot) \text{ if } \neg\bar{\sigma}(\{A, G_j\}) \end{cases}, \quad (4)$$

where $0 \leq j \leq |\mathcal{G}| - 1$, $0 \leq k \leq b - 1$, $\theta = \sigma(\{A, G_j\})$, and $I_{\mathcal{G}}(x)$ returns the index of x in $\mathcal{G}$. If clause head A and ground atom G_j are unifiable, then we put the index of subgoal $B_k\theta$ into the tensor (line 1 in Eq. 4). If the clause has fewer body atoms than the longest clause in $\mathcal{C}$, we fill the gap with the index of $\top$ (line 2 in Eq. 4). If clause head A and ground atom G_j are not unifiable, then we place the index of $\bot$ (line 3 in Eq. 4).

Example 5 Let $C_0 = e(x), C_1 = e(s^2(x)) \leftarrow e(x)$ and $\mathcal{G} = \{\bot, \top, e(0), e(s(0)), e(s^2(0)), e(s^4(0))\}$. Then the following table shows tensor $\mathbf{X}$:

j	0	1	2	3	4	5
$\mathcal{G}$	$\bot$	$\top$	$e(0)$	$e(s(0))$	$e(s^2(0))$	$e(s^4(0))$
$\mathbf{X}[0,j]$	[0]	[1]	[1]	[1]	[1]	[1]
$\mathbf{X}[1,j]$	[0]	[1]	[0]	[0]	[2]	[4]

For example, $\mathbf{X}[1,4] = [2]$ because clause C_1 entails $e(s^2(0))$ with substitution $\theta = \{x = 0\}$. Then subgoal

$e(x)\theta = e(0)$, which has index 2. Clause C_0 does not have a body atom, and so the body is filled by $\top$, which has index 1.

Valuation Valuation vector $\mathbf{v}_t \in \mathbb{R}^{|\mathcal{G}|}$ maps each ground atom into a continuous value at each time step t. The background knowledge is compiled into $\mathbf{v}_0$:

$$\mathbf{v}_0[j] = \mathbf{f}_{convert}(\mathcal{B})[j] = \begin{cases} 1 \ (G_j \in \mathcal{B} \vee G_j = \top) \\ 0 \ (\text{otherwise}) \end{cases}. \quad (5)$$

The differentiable inference function is performed based on valuation vectors. To compute the T-step forward-chaining inference, we compute the sequence of valuation vectors $\mathbf{v}_0, \ldots, \mathbf{v}_T$ in the differentiable inference process.

Clause Weights We assign weights to softly compose the logic programs as follows: (i) We fix the target programs' size as m, i.e., where we try to find a logic program with m clauses. (ii) We introduce $|\mathcal{C}|$-dim weights $\mathcal{W} = \{\mathbf{w}_1, \ldots, \mathbf{w}_m\}$. (iii) We take the softmax of each weight vector $\mathbf{w}_l \in \mathcal{W}$ and softly choose m clauses to compose the logic program. As a probabilistic interpretation, we define a probability distribution $p(x_i^l)$, where x_i^l is a probabilistic variable representing clause C_i is the l-th component of the target program.

In ∂ILP, the weights are assigned to each pair of clauses by assuming all programs are composed of pairs of clauses for each predicate. In our method, we assign several weights to each clause and softly choose each clause. Our approach enables the solver to deal with complex programs that consist of several clauses with identical predicates.

Differentiable Inference We compose a differentiable function, called an *infer function*, that performs forward-chaining inference. The inference result is obtained:

$$\mathbf{v}_T = \mathbf{f}_{infer}(\mathbf{X}, \mathbf{v}_0, \mathcal{W}, T), \quad (6)$$

where $\mathbf{f}_{infer}$ is the infer function, $\mathbf{X}$ is the index tensor, $\mathbf{v}_0$ is the initial valuation vector, $\mathcal{W}$ is the set of weight vectors, and T is the time step.

The infer function is computed as follows. First, each clause $C_i \in \mathcal{C}$ is compiled into a function $\mathbf{c}_i : \mathbb{R}^{|\mathcal{G}|} \to \mathbb{R}^{|\mathcal{G}|}$:

$$\mathbf{c}_i(\mathbf{v}_t)[j] = \prod_k \mathbf{gather}(\mathbf{v}_t, \mathbf{X}[i])[j,k], \quad (7)$$

where function $\mathbf{gather} : \mathbb{R}^{|\mathcal{G}|} \times \mathbb{N}^{|\mathcal{G}| \times b} \to \mathbb{R}^{|\mathcal{G}| \times b}$ is:

$$\mathbf{gather}(\mathbf{a}, \mathbf{B})[j,k] = \mathbf{a}[\mathbf{B}[j,k]]. \quad (8)$$

The $\mathbf{gather}$ function replaces the indexes of the ground atoms by the current valuation values. To take logical *and* across the subgoals in the body, we take the product across dimension 1.

Next we take the weighted sum of the clause function using $\mathbf{w}_l \in \mathcal{W}$:

$$\mathbf{h}_l(\mathbf{v}_t) = \sum_i \mathbf{softmax}(\mathbf{w}_l)[i] \cdot \mathbf{c}_i(\mathbf{v}_t), \quad (9)$$

where $\mathbf{softmax}(\mathbf{x})[i] = \frac{\exp(\mathbf{x}[i])}{\sum_{i'} \exp(\mathbf{x}[i'])}$. Note that $\mathbf{softmax}(\mathbf{w}_l)[i]$ is interpreted as a probability that $C_i \in \mathcal{C}$ is the l-th component of the target program.

Then we compute the forward-chaining inference using clauses $\mathcal{C}$ and weights $\mathcal{W}$:

$$\mathbf{r}(\mathbf{v}_t) = \mathbf{softor}^\gamma\left(\mathbf{h}_1(\mathbf{v}_t), \ldots, \mathbf{h}_m(\mathbf{v}_t)\right), \quad (10)$$

where $\mathbf{softor}^\gamma$ is a smooth logical *or* function on the valuation vectors:

$$\mathbf{softor}^\gamma(\mathbf{x}_1, \ldots, \mathbf{x}_m)[j] = \gamma \log \sum_l e^{\mathbf{x}_l[j]/\gamma}, \quad (11)$$

where $\gamma > 0$ is a smooth parameter. Taking logical *or* softly for the valuation vectors corresponds to the fact that a logic program is generally represented as a conjunction of clauses.

Finally, we perform T-step inference by iteratively amalgamating the results:

$$\mathbf{v}_{t+1} = \mathbf{softor}^\gamma\left(\mathbf{v}_t, \mathbf{r}(\mathbf{v}_t)\right). \quad (12)$$

Infer function $\mathbf{f}_{infer}(\mathbf{X}, \mathbf{v}_0, \mathcal{W}, T)$ returns $\mathbf{v}_T$.

Learn Target Program

Let $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-, \mathcal{B}, \mathcal{L})$. We generate pairs of atoms and labels as:

$$\mathcal{Y} = \{(E, 1) \mid E \in \mathcal{E}^+\} \cup \{(E, 0) \mid E \in \mathcal{E}^-\}. \quad (13)$$

Each pair (E, y) represents whether atom E is positive or negative. We compute the conditional probability of label y of atom E:

$$p(y \mid E, \mathcal{Q}, \mathcal{C}, \mathcal{W}, T) = \mathbf{f}_{infer}(\mathbf{X}, \mathbf{v}_0, \mathcal{W}, T)[I_{\mathcal{G}}(E)], \quad (14)$$

where $\mathcal{C} = f_{beam\_search}(\mathcal{C}_0, \mathcal{Q}, N_{beam}, T_{beam})$, $\mathcal{G} = f_{enumerate}(\mathcal{C}, \mathcal{Q}, T)$, $\mathbf{v}_0 = \mathbf{f}_{convert}(\mathcal{B})$, $\mathbf{X}$ is the index tensor, and $I_{\mathcal{G}}(x)$ returns the index of x in $\mathcal{G}$. Here f_{beam_search} is the clause generation function following Algorithm 1, $f_{enumerate}$ is the fact enumeration function following Algorithm 2, $\mathcal{W}$ is the set of weights, and T is the time step for the infer function.

We solve ILP problem $\mathcal{Q}$ by minimizing cross-entropy loss, defined as:

$$loss = -\mathbb{E}_{(E,y)\sim\mathcal{Y}}[y \log p(y \mid E, \mathcal{Q}, \mathcal{C}, \mathcal{W}, T) + (1-y)\log(1 - p(y \mid E, \mathcal{Q}, \mathcal{C}, \mathcal{W}, T))]. \quad (15)$$

Experiments

In this section, we experimentally support the following claims: (1) Our enumeration algorithm yields a reasonable number of ground atoms. (2) Our clause generation algorithm improves the performance of differentiable program searching. (3) Our soft program composition is efficient in terms of memory and computation costs. (4) Our framework learns logic programs successfully from noisy and structured examples, which are outside the scope of both ∂ILP and standard ILP approaches.

We performed our experiments[1] on several standard ILP tasks with structured examples, partially adopted from

[1]The source code of all experiments will be available at https://github.com/hkrsnd/dilp-st

Shapiro and Caferra (Shapiro 1983; Caferra 2013). Through all the tasks, sets of variables were consistently fixed, i.e., $\mathcal{V} = \{x, y, z, v, w\}$.

Member The task is to learn the membership function for lists. The language is given as $\mathcal{P} = \{mem/2\}$, $\mathcal{F} = \{f/2\}$, $\mathcal{A} = \{a, b, c, *\}$. The initial clause is $\mathcal{C}_0 = \{mem(x, y)\}$. The problem is briefly described:

$$\mathcal{E}^+ = \{mem(a, [a, c]), mem(a, [b, a]), \ldots\},$$
$$\mathcal{E}^- = \{mem(c, [b, a]), mem(c, [a]), \ldots\},$$
$$\mathcal{B} = \{mem(a, [a]), mem(b, [b]), mem(c, [c])\}.$$

Plus The task is to learn the plus operation for natural numbers. The language is given as $\mathcal{P} = \{plus/3\}$, $\mathcal{F} = \{s/1\}$, $\mathcal{A} = \{0\}$. The initial clause is $\mathcal{C}_0 = \{plus(x, y, z)\}$. The problem is briefly described:

$$\mathcal{E}^+ = \{plus(s(0), 0, s(0)), plus(s^5(0), s^3(0), s^8(0)), \ldots\},$$
$$\mathcal{E}^- = \{plus(s(0), s^2(0), 0), plus(0, s^2(0), s^4(0)), \ldots\},$$
$$\mathcal{B} = \{plus(0, 0, 0)\}.$$

Append The task is to learn the append function for lists. The language is given as $\mathcal{P} = \{app/3\}$, $\mathcal{F} = \{f/2\}$, $\mathcal{A} = \{a, b, c, *\}$. The initial clause is $\mathcal{C}_0 = \{app(x, y, z)\}$. The problem is briefly described:

$$\mathcal{E}^+ = \{app([c], [], [c]), app([a, a, b], [b, c], [a, a, b, b, c]), \ldots\},$$
$$\mathcal{E}^- = \{app([], [a, a], [a, a, b]), app([b], [], [c]), \ldots\},$$
$$\mathcal{B} = \{app([], [], [])\}.$$

Delete The task is to learn the delete operation for lists. The language is given as $\mathcal{P} = \{del/3\}$, $\mathcal{F} = \{f/2\}$, $\mathcal{A} = \{a, b, c, *\}$. The initial clause is $\mathcal{C}_0 = \{del(x, y, z)\}$. The problem is briefly described:

$$\mathcal{E}^+ = \{del(b, [a, c, b], [a, c]), del(a, [b, a, a], [b, a]), \ldots\},$$
$$\mathcal{E}^- = \{del(c, [c, a, a], [a, b]), del(b, [b], [a]), \ldots\},$$
$$\mathcal{B} = \{del(a, [a], []), del(b, [b], []), del(c, [c], [])\}.$$

Subtree The task is to learn the subsumption relation for binary trees. The language is given as $\mathcal{P} = \{sub/2\}$, $\mathcal{F} = \{f/2\}$, $\mathcal{A} = \{a, b, c\}$. The initial clause is $\mathcal{C}_0 = \{sub(x, y)\}$. The problem is briefly described:

$$\mathcal{E}^+ = \{sub(f(b, b), f(f(f(b, b), f(a, c)), f(a, c))), \ldots\},$$
$$\mathcal{E}^- = \{sub(f(a, a), f(f(c, a), f(a, c))), \ldots\},$$
$$\mathcal{B} = \{sub(a, a), sub(b, b), sub(c, c)\}.$$

In each task, we randomly generate 50 examples for each class. Note that the list objects are represented in a readable form, e.g., term $f(a, f(b, *))$ is represented as $[a, b]$.

Experimental Methods and Results

Hyperparameters To generate clauses, we used several biases for them: (i) the maximum number of bodies, denoted by N_{body}, and (ii) the maximum number of the nests of function symbols, denoted by N_{nest}. In all experiments, we set $N_{body} = 1$ and $N_{nest} = 1$. We set beam size N_{beam}, and beam step T_{beam} is $(N_{beam}, T_{beam}) = (3, 3)$ for the Member task, $(N_{beam}, T_{beam}) = (15, 3)$ for the Subtree task, and $(N_{beam}, T_{beam}) = (10, 5)$ for the other tasks.

Member	Plus	Append	Delete	Subtree
228	1857	2899	2513	2172

Table 1: Number of enumerated ground atoms

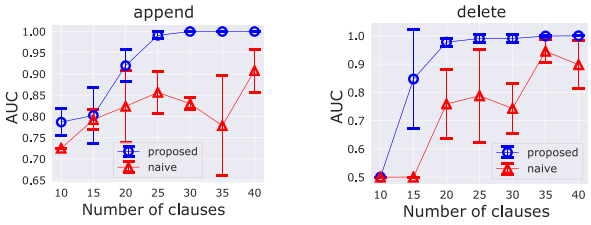

Figure 3: AUC for number of generated clauses

We set target program size m as $m = 2$ for the Member and Delete tasks, $m = 3$ for the Plus and Append tasks, and $m = 4$ for the Subtree task. We set T for the differentiable inference as $T = 8$ for the Plus task and $T = 4$ for the other tasks. We set $\gamma = 10^{-5}$ for the softor function.

We trained our model with the RMSProp optimizer with a learning rate of 0.01 for 3000 epochs. We sampled mini-batches during the optimization, and each mini-batch contained 5% of the training examples chosen randomly for each iteration. The weights were initialized randomly in each trial. We divided the data into 70% training and 30% test. All experiments were performed on a desktop computer using its GPU[2].

Experiment 1 To support claim 1, we show the number of enumerated ground atoms for training data in each dataset in Table 1. Our enumeration algorithm yielded a reasonable number of ground atoms in each dataset. The ∂ILP approach is infeasible in our setting because, although it considers all the ground atoms generated in the language, an infinite number of them can be generated with function symbols.

Experiment 2 To support claim 2, we compared *two* clause generation algorithms: (i) generation by beam searching and refinement and (ii) naive generation without beam searching. In setting (ii), we generated clauses without evaluation by examples. Like ∂ILP, it did not use the given examples during clause generation. We set a number of clauses, denoted by N_{clause}. The generation stopped when the number of generated clauses exceeded N_{clause}. We performed classification with different N_{clause}. We changed the value from 10 to 40 by increments of 10 and ran the experiments 5-times with random-weight initialization.

Figure 3 shows the AUC for the Append and Delete tasks. In each task, our approach achieved AUC scores of 1.0 with fewer clauses. These results show that our clause generation algorithm improved the differentiable solver, i.e., yielded an efficient search space.

Experiment 3 To support claim 3, we compared 2 different approaches for the infer function: (i) multiple weights and softor approach (proposed here) and (ii) 2-d weights

[2]CPU: Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60 GHz, GPU: GeForce 1080Ti 11 GB, RAM: 64 GB

	Parameters		Runtime [s]	
	Proposed	Pair	Proposed	Pair
Member	24	144	**0.015**	0.12
Plus	120	1600	**0.03**	6.91
Append	150	2500	**0.09**	5.18
Delete	150	2500	**0.06**	5.4
Subtree	80	400	**0.039**	1.11

Table 2: Number of parameters and mean runtime in learning steps

for pairs of clauses. Setting (ii) is a ∂ILP approach, which defines a probability distribution over the pairs of clauses $\mathcal{C}$, i.e., we assigned weights in the form of a 2-d matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{C}|\times|\mathcal{C}|}$. We compared the number of parameters and mean runtimes for each step of the gradient descent.

Table 2 shows our results. In each dataset, the proposed approach had fewer parameters. Moreover, the mean runtime of the gradient descent was much shorter than with the pairing approach. These results show that our approach was efficient in terms of memory and computation costs.

Experiment 4 To support claim 4, we evaluated our approach by changing the proportion of the mislabeled training data. First, we generated training examples. Then we flipped the label of examples to make noise according to the proportion. We changed the proportion of mislabeled data from 0.0 to 0.5 by increments of 0.05. We ran the experiments 5-times with random-weight initialization.

Figure 4 shows the mean-squared test error for the proportion of mislabeled training data in the Member and Subtree tasks. In each task, the test error increased gradually as the noise proportion increased. Moreover, our method achieved test error less than 0.05 with 10% mislabeled training data in both tasks. This shows that our approach was robust to noise, i.e., it found a functional theory even if there were mislabeled data. Note that standard ILP approaches fail to find a theory when there are mislabeled data.

We show an example of the obtained programs in Table 3. The clauses for lists are represented in a readable form, e.g., term $f(x, y)$ is represented as $[x|y]$. In the Plus task, the last clause represents the plus operation considering commutativity for natural numbers. Also the last clause in the Append task can be interpreted clearly. If v is obtained by appending y to z, then the result of appending y with head x to z is obtained just by v with head x. Our framework learned structured knowledge from structured examples beyond relational logic.

Conclusion

We proposed a new differentiable inductive logic programming framework that deals with complex logic programs with function symbols that yield readable outputs for structured data. To establish our framework, we proposed $three$ main contributions. First, we proposed a clause generation algorithm that uses beam searching with refinement. Second, we proposed an enumeration algorithm for ground atoms. Third, we proposed a soft program composition approach using multiple weights and the softor function.

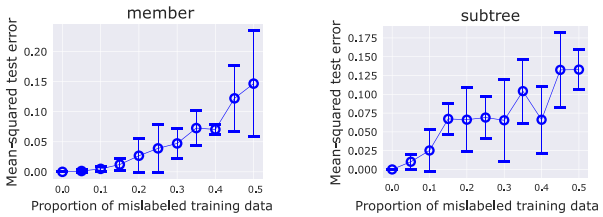

Figure 4: Mean-squared test error as proportion of mislabeled training data

Problem	Learned logic program
Member	$mem(x, [y\|z]) \leftarrow mem(x, z)$ $mem(x, [x\|y])$
Plus	$plus(0, x, x)$ $plus(x, s(y), s(z)) \leftarrow plus(x, y, z)$ $plus(s(x), y, s(z)) \leftarrow plus(y, x, z)$
Append	$app([], x, x)$ $app(x, [], x)$ $app([x\|y], z, [x\|v]) \leftarrow app(y, z, v)$
Delete	$del(x, [x\|y], y)$ $del(x, [y\|z], [y\|v]) \leftarrow del(x, z, v)$
Subtree	$sub(f(x, y), f(x, y))$ $sub(x, f(y, z)) \leftarrow sub(x, z)$ $sub(x, f(y, z)) \leftarrow sub(x, y)$ $sub(x, f(y, x))$

Table 3: Learned logic programs in standard ILP tasks

In our experiments, we showed: (i) our enumeration algorithm yields a reasonable number of ground atoms, (ii) our clause generation algorithm improves the performance of differentiable program searching, (iii) our soft program composition is efficient in memory and computation costs, and (iv) our framework learns logic programs successfully from noisy and structured examples, which are outside the scope of both ∂ILP and standard ILP approaches.

One major limitation of our framework is its scalability for large-scale programs. A high-quality search space is necessary to deal with more expressive programs, such as sorting. Further research could tackle this problem by incorporating such declarative bias (Claire et al. 1996; De Raedt 2012) as mode declarations (Muggleton 1991) and metarules (Cropper, Tamaddoni-Nezhad, and Muggleton 2015) to manage the search space.

To the best of our knowledge, this is the first work that incorporates symbolic methods, such as refinement, with a differentiable ILP approach. We believe that our work will trigger future work to combine the best of both the symbolic and subsymbolic worlds.

Acknowledgments

This work was partly supported by JSPS KAKENHI Grant Number 17K19973.

References

Arikawa, S.; Shinohara, T.; and Yamamoto, A. 1992. Learning elementary formal systems. In *Theoretical Computer Science*, volume 95, 97–113.

Bellodi, E.; and Riguzzi, F. 2015. Structure learning of probabilistic logic programs by searching the clause space. *Theory Pract. Log. Program.* 15(2): 169–212.

Caferra, R. 2013. *Logic for Computer Science and Artificial Intelligence*. Wiley.

Chikara, N.; Koshimura, M.; Fujita, H.; and Hasegawa, R. 2015. Inductive logic programming using a MaxSAT solver. In *25th International Conference on Inductive Logic Programming (ILP 2015)*.

Claire, N.; Céline, R.; Hilde, A.; Francesco, B.; and Birgit, T. 1996. Declarative bias in ILP. *Advances in inductive logic programming* 32: 82—103.

Cohen, W. W.; Yang, F.; and Mazaitis, K. 2020. TensorLog: A Probabilistic Database Implemented Using Deep-Learning Infrastructure. *J. Artif. Intell. Res. (JAIR)* 67: 285–325.

Cropper, A.; Tamaddoni-Nezhad, A.; and Muggleton, S. H. 2015. Meta-Interpretive Learning of Data Transformation Programs. In *25th International Conference on Inductive Logic Programming (ILP 2015)*, volume 9575, 46–59.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.* 33(3): 374–425.

d'Avila Garcez, A. S.; Gori, M.; Lamb, L. C.; Serafini, L.; Spranger, M.; and Tran, S. N. 2019. Neural-symbolic Computing: An Effective Methodology for Principled Integration of Machine Learning and Reasoning. *FLAP* 6(4): 611–632.

De Raedt, L. 2012. Declarative Modeling for Machine Learning and Data Mining. In *24th International Conference on Algorithmic Learning Theory (ALT 2013)*, 12–12.

De Raedt, L.; Dumancic, S.; Manhaeve, R.; and Marra, G. 2020. From Statistical Relational to Neuro-Symbolic Artificial Intelligence. In *29th International Joint Conference on Artificial Intelligence (IJCAI 2020)*, 4943–4950.

De Raedt, L.; and Kersting, K. 2004. Probabilistic Inductive Logic Programming. In *15th International Conference on Algorithmic Learning Theory (ALT 2004)*, 19–36.

De Raedt, L.; Kersting, K.; Natarajan, S.; and Poole, D. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Evans, R.; and Grefenstette, E. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Intell. Res. (JAIR)* 61: 1–64.

Fredouille, D. C.; Bryant, C. H.; Jayawickreme, C. K.; Jupe, S.; and Topp, S. 2007. An ILP Refinement Operator for Biological Grammar Learning. In *16th International Conference on Inductive Logic Programming (ILP 2006)*, 214–228.

Kok, S.; and Domingos, P. 2005. Learning the Structure of Markov Logic Networks. In *22th International Conference on Machine Learning (ICML 2005)*, 441–448.

Lloyd, J. W. 2003. *Logic for Learning*. Springer-Verlag Berlin Heidelberg.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. DeepProbLog: Neural Probabilistic Logic Programming. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, 3749–3759.

Marra, G.; Diligenti, M.; Giannini, F.; Gori, M.; and Maggini, M. 2020. Relational Neural Machines. In *24th European Conference on Artificial Intelligence (ECAI 2020)*.

Muggleton, S. 1991. Inductive logic programming. *New Generation Computing* 8(4): 295–318.

Nguembang Fadja, A.; and Riguzzi, F. 2019. Lifted discriminative learning of probabilistic logic programs. *Machine Learning* 108(7): 1111–1135.

Nienhuys-Cheng, S.-H.; Wolf, R. d.; Siekmann, J.; and Carbonell, J. G. 1997. *Foundations of Inductive Logic Programming*. Springer-Verlag.

Plotkin, G. 1971. A further note on inductive generalization. In *Machine Intelligence*, volume 6. Edinburgh University Press.

Riegel, R.; Gray, A. G.; Luus, F. P. S.; Khan, N.; Makondo, N.; Akhalwaya, I. Y.; Qian, H.; Fagin, R.; Barahona, F.; Sharma, U.; Ikbal, S.; Karanam, H.; Neelam, S.; Likhyani, A.; and Srivastava, S. K. 2020. Logical Neural Networks. *CoRR* abs/2006.13155.

Rocktäschel, T.; and Riedel, S. 2017. End-to-end Differentiable Proving. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, 3788–3800.

Shapiro, E. Y. 1983. *Algorithmic Program DeBugging*. MIT Press.

Shindo, H.; Nishino, M.; and Yamamoto, A. 2018. Using Binary Decision Diagrams to Enumerate Inductive Logic Programming Solutions. In *28th International Conference on Inductive Logic Programming (ILP 2018)*, 52–67.

Si, X.; Raghothaman, M.; Heo, K.; and Naik, M. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *28th International Joint Conference on Artificial Intelligence, (IJCAI 2019)*, 6117–6124.

Šourek, G.; Aschenbrenner, V.; Železný, F.; Schockaert, S.; and Kuželka, O. 2018. Lifted Relational Neural Networks: Efficient Learning of Latent Relational Structures. *J. Artif. Int. Res. (JAIR)* 62(1): 69–100.

Yang, F.; Yang, Z.; and Cohen, W. W. 2017. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, 2316–2325.