Synthesizing Datalog Programs Using Numerical Relaxation

Xujie Si[*], **Mukund Raghothaman**[*], **Kihong Heo** and **Mayur Naik**
University of Pennsylvania
{xsi, rmukund, kheo, mhnaik}@cis.upenn.edu

Abstract

The problem of learning logical rules from examples arises in diverse fields, including program synthesis, logic programming, and machine learning. Existing approaches either involve solving computationally difficult combinatorial problems, or performing parameter estimation in complex statistical models.

In this paper, we present DIFFLOG, a technique to extend the logic programming language Datalog to the continuous setting. By attaching real-valued weights to individual rules of a Datalog program, we naturally associate numerical values with individual conclusions of the program. Analogous to the strategy of numerical relaxation in optimization problems, we can now first determine the rule weights which cause the best agreement between the training labels and the induced values of output tuples, and subsequently recover the classical discrete-valued target program from the continuous optimum.

We evaluate DIFFLOG on a suite of 34 benchmark problems from recent literature in knowledge discovery, formal verification, and database query-by-example, and demonstrate significant improvements in learning complex programs with recursive rules, invented predicates, and relations of arbitrary arity.

1 Introduction

As a result of its rich expressive power and efficient implementations, the logic programming language Datalog has witnessed applications in diverse domains such as bioinformatics [Seo, 2018], big-data analytics [Shkapsky *et al.*, 2016], robotics [Poole, 1995], networking [Loo *et al.*, 2006], and formal verification [Bravenboer and Smaragdakis, 2009]. Users on the other hand are often unfamiliar with logic programming. The programming-by-example (PBE) paradigm aims to bridge this gap by providing an intuitive interface for non-expert users [Gulwani, 2011].

Typically, a PBE system is given a set of input tuples and sets of desirable and undesirable output tuples. The central computational problem is that of synthesizing a Datalog program, i.e., a set of logical inference rules which produces, from the input tuples, a set of conclusions which is compatible with the output specification. Previous approaches to this problem focus on optimizing the combinatorial exploration of the search space. For example, ALPS maintains a small set of syntactically most-general and most-specific candidate programs [Si *et al.*, 2018], Zaatar encodes the derivation of output tuples as a SAT formula for subsequent solving by a constraint solver [Albarghouthi *et al.*, 2017], and inductive logic programming (ILP) systems employ sophisticated pruning algorithms based on ideas such as inverse entailment [Muggleton, 1995]. Given the computational complexity of the search problem, however, these systems are hindered by large or difficult problem instances. Furthermore, these systems have difficulty coping with minor user errors or noise in the training data.

In this paper, we take a fundamentally different approach to the problem of synthesizing Datalog programs. Inspired by the success of numerical methods in machine learning and other large scale optimization problems, and of the strategy of relaxation in solving combinatorial problems such as integer linear programming, we extend the classical discrete semantics of Datalog to a continuous setting named DIFFLOG, where each rule is annotated with a real-valued weight, and the program computes a numerical value for each output tuple. This step can be viewed as an instantiation of the general K-relation framework for database provenance [Green *et al.*, 2007] with the Viterbi semiring being chosen as the underlying space K of provenance tokens. We then formalize the program synthesis problem as that of selecting a subset of target rules from a large set of candidate rules, and thereby uniformly capture various methods of inducing syntactic bias, including syntax-guided synthesis (SyGuS) [Alur *et al.*, 2015], and template rules in meta-interpretive learning [Muggleton *et al.*, 2015].

The synthesis problem thus reduces to that of finding the values of the rule weights which result in the best agreement between the computed values of the output tuples and their specified values (1 for desirable and 0 for undesirable tuples). The fundamental NP-hardness of the underlying decision problem manifests as a complex search surface, with local minima and saddle points. To overcome these challenges, we devise a hybrid optimization algorithm which combines Newton's root-finding method with periodic invocations of a simulated annealing search. Finally, when the optimum value is reached, connections between the semantics of DIFFLOG and Datalog

[*]Co-first authors.

enable the recovery of a classical discrete-valued Datalog program from the continuous-valued optimum produced by the optimization algorithm.

A particularly appealing aspect of relaxation-based synthesis is the randomness caused by the choice of the starting position and of subsequent Monte Carlo iterations. This manifests both as a variety of different solutions to the same problem, and as a variation in running times. Running many search instances in parallel therefore enables stochastic speedup of the synthesis process, and allows us to leverage compute clusters in a way that is fundamentally impossible with deterministic approaches. We have implemented DIFFLOG and evaluate it on a suite of 34 benchmark programs from recent literature. We demonstrate significant improvements over the state-of-the-art, even while synthesizing complex programs with recursion, invented predicates, and relations of arbitrary arity.

Contributions. Our work makes the following contributions:

1. A formulation of the Datalog synthesis problem as that of selecting a set of desired rules. This formalism generalizes syntax-guided query synthesis and meta-rule guided search.
2. A fundamentally new approach to solving rule selection by numerically minimizing the difference between the weighted set of candidate rules and the reference output.
3. An extension of Datalog which also associates output tuples with numerical weights, and which is a continuous refinement of the classical semantics.
4. Experiments showing state-of-the-art performance on a suite of diverse benchmark programs from recent literature.

2 Related Work

Weighted logical inference. The idea of extending logical inference with weights has been studied by the community in statistical relational learning. [Shapiro, 1983] proposes *quantitative logic programming* to measure the uncertainty of expert systems by associating logical rules with uncertainty scores. Markov Logic Networks [Richardson and Domingos, 2006; Kok and Domingos, 2005] view a first order formula as a template for generating a Markov random field, where the weight attached to the formula specifies the likelihood of its grounded clauses. ProbLog [De Raedt *et al.*, 2007] extends logic programing languages with probabilistic rules and reduces the inference problem to weighted model counting. DeepProbLog [Manhaeve *et al.*, 2018] further extends ProbLog with neural predicates (e.g., input data which can be images). In another direction, aProbLog [Kimmig *et al.*, 2011; Kimmig *et al.*, 2017] generalizes ProbLog by associating logical rules with elements from a semiring, instead of just probability values. These frameworks could conceivably serve as the underlying inference engine of our framework but we use the Viterbi semiring because: (*a*) inference in these frameworks is #P-complete and only requires polynomial time in the Viterbi semiring; and (*b*) automatic differentiation is either inefficient or simply not available.

Structure learning for probabilistic logics. Weight learning has also been used as a means to structure learning [Muggleton, 1996; Wang *et al.*, 2014; Embar *et al.*, 2018]; however, our work has two significant differences: First, the values we assign to tuples do not have natural interpretations as probabilities, so that exact inference can be performed just as efficiently as solving Datalog programs. Furthermore, while the search trajectory itself proceeds through smoothed programs with non-zero loss, our termination criterion ensures that the final result is still a classical Datalog program which is consistent with the provided examples.

Inductive logic programming (ILP). The Datalog synthesis problem can also be seen as an instance of the classic ILP problem. [Cohen and Page, 1995] show that learning a single rule that is consistent with labelled examples is NP-hard: this is similar to our motivating result in Theorem 2, where we demonstrate NP-hardness even if candidate rules are explicitly specified. Metagol [Muggleton *et al.*, 2015] supports higher-order dyadic Datalog synthesis but the synthesized program can only consist of binary relations. Metagol is built on top of Prolog which makes the system very expressive but also introduces difficult issues with non-terminating programs. In contrast, by performing a bidirectional search based on query-by-committee and building on top of the Z3 fixpoint engine, ALPS [Si *et al.*, 2018] exhibits significantly greater scalability while synthesizing Datalog programs. Recent works such as NeuralLP [Yang *et al.*, 2017] and ∂ILP [Evans and Grefenstette, 2018] cast logic program synthesis as a differentiable end-to-end learning problem and model relation joins as a form of matrix multiplication, which also limits them to binary relations. NTP [Rocktäschel and Riedel, 2017] constructs a neural network as a learnable proof (or derivation) for each output tuple up to a predefined depth (e.g. ≤ 2) with a few (e.g. ≤ 4) templates, where the neural network could be exponentially large when either the depth or the number of templates grows. The predefined depth and a small number of templates could significantly limit the class of learned programs. Our work seeks to synthesize Datalog programs consisting of relations of arbitrary arity and support rich features like recursion and predicate invention.

MCMC methods for program synthesis. Markov chain Monte-Carlo (MCMC) methods have also been used for program synthesis. For example, in STOKE, [Schkufza *et al.*, 2016] apply the Metropolis-Hastings algorithm to synthesize efficient loop free programs. Similarly, [Liang *et al.*, 2010] show that program transformations can be efficiently learned from demonstrations by MCMC inference.

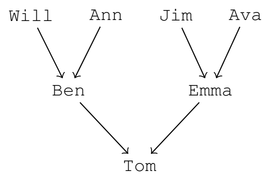
3 The Datalog Synthesis Problem

In this section, we concretely describe the Datalog synthesis problem, and establish some basic complexity results. We use the family tree shown in Figure 1 as a running example. In Section 3.1, we briefly describe how one may compute $\mathsf{samegen}(x, y)$ from $\mathsf{parent}(x, y)$ using a Datalog program. In Section 3.2, we formalize the query synthesis problem as that of rule selection.

3.1 Overview of Datalog

The set of tuples inhabiting relation $\mathsf{samegen}(x, y)$ can be computed using the following pair of *inference rules*, r_1 and r_2:

r_1: $\mathsf{samegen}(x, y) \coloneq \mathsf{parent}(x, z), \mathsf{parent}(y, z).$



(a)

Input tuples (EDB)
parent (Will, Ben)
parent (Ann, Ben)
parent (Jim, Emma)
parent (Ava, Emma)
parent (Ben, Tom)
parent (Emma, Tom)

(b)

Output tuples (IDB)
samegen (Ben, Emma)
samegen (Ann, Will)
samegen (Jim, Ann)
...

(c)

Figure 1: Example of a family tree (a), and its representation as a set of input tuples (b). An edge from x to y indicates that x is a parent of y, and is represented symbolically as the tuple $\mathsf{parent}(x, y)$. The user wishes to realize the relation $\mathsf{samegen}(x, y)$, indicating the fact that x and y occur are from the same generation of the family (c).

r_2: $\mathsf{samegen}(x, u) :- \mathsf{parent}(x, y), \mathsf{parent}(u, v), \mathsf{samegen}(y, v)$.

Rule r_1 describes the fact that for all persons x, y, and z, if both x and y are parents of z, then x and y occur at the same level of the family tree. Informally, this rule forms the base of the inductive definition. Rule r_2 forms the inductive step of the definition, and provides that x and u occur in the same generation whenever they have children y and v who themselves occur in the same generation.

By convention, the relations which are explicitly provided as part of the input are called the EDB, $\mathcal{I} = \{\mathsf{parent}\}$, and those which need to be computed as the output of the program are called the IDB, $\mathcal{O} = \{\mathsf{samegen}\}$. To evaluate this program, one starts with the set of input tuples, and repeatedly applies rules r_1 and r_2 to derive new output tuples. Note that because of the appearance of the literal $\mathsf{samegen}(y, v)$ on the right side of rule r_2, discovering a single output tuple may recursively result in the further discovery of additional output tuples. The derivation process ends when no additional output tuples can be derived, i.e., when the set of conclusions reaches a *fixpoint*.

More generally, we assume a collection of *relations*, $\{P, Q, \dots\}$. Each relation P has an arity $k \in \mathbb{N}$, and is a set of *tuples*, each of which is of the form $P(c_1, c_2, \dots, c_k)$, for some *constants* $c_1, c_2, \dots, c_k$. The Datalog program is a collection of rules, where each rule r is of the form:

$$P_h(\boldsymbol{u}_h) :- P_1(\boldsymbol{u}_1), P_2(\boldsymbol{u}_2), \dots, P_k(\boldsymbol{u}_k),$$

where P_h is an output relation, and $\boldsymbol{u}_h$, $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, ..., $\boldsymbol{u}_k$ are vectors of *variables* of appropriate length. The variables $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, ..., $\boldsymbol{u}_k$, $\boldsymbol{u}_h$ appearing in the rule are implicitly universally quantified, and instantiating them with appropriate constants $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, ..., $\boldsymbol{v}_k$, $\boldsymbol{v}_h$ yields a grounded constraint g of the form $P_1(\boldsymbol{v}_1) \wedge P_2(\boldsymbol{v}_2) \wedge \cdots \wedge P_k(\boldsymbol{v}_k) \implies P_h(\boldsymbol{v}_h)$: *"If all of the antecedent tuples $A_g = \{P_1(\boldsymbol{v}_1), P_2(\boldsymbol{v}_2), \dots, P_k(\boldsymbol{v}_k)\}$ are derivable, then the conclusion $c_g = P_h(\boldsymbol{v}_h)$ is also derivable."*

3.2 Synthesis as Rule Selection

The input-output examples, I, O_+, and O_-. Instead of explicitly providing rules r_1 and r_2, the user provides an example instance of the EDB I, and labels a few tuples of the output relation as "desirable" or "undesirable" respectively:

$$O_+ = \{\mathsf{samegen}(\mathtt{Ann}, \mathtt{Jim})\}, \text{ and}$$

$$O_- = \{\mathsf{samegen}(\mathtt{Ava}, \mathtt{Tom}), \mathsf{samegen}(\mathtt{Jim}, \mathtt{Emma})\},$$

indicating that Ann and Jim are from the same generation, but Ava and Tom and Jim and Emma are not. Note that the user is free to label as many potential output tuples as they wish, and the provided labels $O_+ \cup O_-$ need not be exhaustive. The goal of the program synthesizer is to find a set of rules R_s which produce all of the desired output tuples, i.e., $O_+ \subseteq R_s(I)$, and none of the undesired tuples, i.e., $O_- \cap R_s(I) = \emptyset$.

The set of candidate rules, R. The user often possesses additional information about the problem instance and the concept being targeted. This information can be provided to the synthesizer through various forms of *bias*, which direct the search towards desired parts of the search space. A particularly common form in the recent literature on program synthesis is syntactic: for example, SyGuS requires a description of the space of potential solution programs as a context-free grammar [Alur *et al.*, 2015], and recent ILP systems such as Metagol [Muggleton *et al.*, 2015] require the user to provide a set of higher-order rule templates ("*metarules*") and order constraints over predicates and variables that appear in clauses. In this paper, we assume that the user has provided a large set of candidate rules R and that the target concept R_s is a subset of these rules: $R_s \subseteq R$.

These candidate rules can express various patterns that could conceivably discharge the problem instance. For example, R can include the candidate rule r_s, "$\mathsf{samegen}(x, y) :- \mathsf{samegen}(y, x)$", which indicates that the output relation is symmetric, and the candidate rule r_t, "$\mathsf{samegen}(x, z) :- \mathsf{samegen}(x, y), \mathsf{samegen}(y, z)$", which indicates that the relation is transitive. Note that the assumption of the candidate rule set R uniformly subsumes many previous forms of syntactic bias, including those in SyGuS and Metagol.

Also note that R can often be automatically populated: In our experiments in Section 6, we automatically generate R using the approach introduced by ALPS [Si *et al.*, 2018]. We start with seed rules that follow a simple chain pattern (e.g., "$P_1(x_1, x_4) :- P_2(x_1, x_2), P_3(x_2, x_3), P_4(x_3, x_4)$"), and repeatedly augment R with simple edits to the variables, predicates, and literals of current candidate rules. The candidate rules thus generated exhibit complex patterns, including recursion, and contain literals of arbitrary arity. Furthermore, any conceivable Datalog rule can be produced with a sufficiently large augmentation distance.

Problem 1 (Rule Selection)**.** *Let the following be given: (a) a set of input relations, $\mathcal{I}$ and output relations, $\mathcal{O}$, (b) the set of input tuples I, (c) a set of positive output tuples O_+, (d) a set of negative output tuples O_-, and (e) a set of candidate rules R which map the input relations $\mathcal{I}$ to the output relations $\mathcal{O}$. Find a set of target rules $R_s \subseteq R$ such that:*

$$O_+ \subseteq R_s(I), \textit{ and } O_- \cap R_s(I) = \emptyset.$$

Finally, we note that the rule selection problem is NP-hard: this is because multiple rules in the target program R_s may interact in non-compositional ways. The proof proceeds through a straightforward encoding of the satisfiability of a 3-CNF formula, and is provided in the Appendix.

Theorem 2. *Determining whether an instance of the rule selection problem, $(\mathcal{I}, \mathcal{O}, I, O_+, O_-, R)$, admits a solution is NP-hard.*

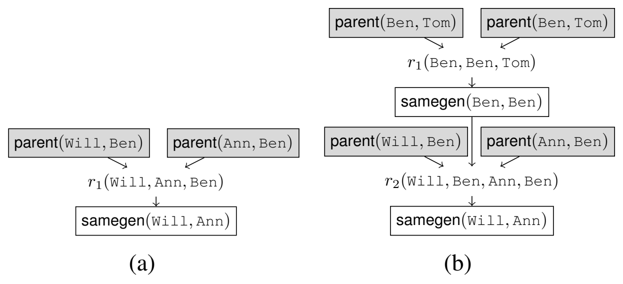


Figure 2: Examples of derivation trees, τ_1 (a) and τ_2 (b) induced by various combinations of candidate rules, applied to the EDB of familial relationships from Figure 1. The input tuples are shaded in grey. We present two derivation trees for the conclusion samegen(Will, Ann) using rules r_1 and r_2 in Section 3.1.

4 A Smoothed Interpretation for Datalog

In this section, we describe the semantics of DIFFLOG, and present an algorithm to evaluate and automatically differentiate this continuous-valued extension.

4.1 Relaxing Rule Selection

The idea motivating DIFFLOG is to generalize the concept of rule selection: instead of a set of binary decisions, we associate each rule r with a numerical weight $w_r \in [0, 1]$. One possible way to visualize these weights is as the extent to which they are present in the current candidate program. The central challenge, which we will now address, is in specifying how the vector of rule weights $\boldsymbol{w}$ determines the numerical values $v_t^{R,I}(\boldsymbol{w})$ for the output tuples t of the program. We will simply write $v_t(\boldsymbol{w})$ when the set of rules R and the set of input tuples I are evident from context.

Every output tuple of a Datalog program is associated with a set of derivation trees, such as those shown in Figure 2. Let r_g be the rule associated with each instantiated clause g that appears in the derivation tree τ. We define the value of τ, $v_\tau(\boldsymbol{w})$, as the product of the weights of all clauses appearing in τ, and the value of an output tuple t as being the supremum of the values of all derivation trees of which it is the conclusion:

$$v_\tau(\boldsymbol{w}) = \prod_{\text{clause } g \in \tau} w_{r_g}, \text{ and} \tag{1}$$

$$v_t(\boldsymbol{w}) = \sup_{\tau \text{ with conclusion } t} v_\tau(\boldsymbol{w}), \tag{2}$$

with the convention that $\sup(\emptyset) = 0$. For example, if $w_{r_1} = 0.8$ and $w_{r_2} = 0.6$, then the weight of the trees τ_1 and τ_2 from Figure 2 are respectively $v_{\tau_1}(\boldsymbol{w}) = w_{r_1} = 0.8$ and $v_{\tau_2}(\boldsymbol{w}) = w_{r_1} w_{r_2} = 0.48$.

Since $0 \leq w_r \leq 1$, it follows that $v_\tau(\boldsymbol{w}) \leq 1$. Also note that a single output tuple may be the conclusion of infinitely many proof trees (see the derivation structure in Figure 3), leading to the deliberate choice of the supremum in Equation 2.

One way to consider Equations 1 and 2 is as replacing the traditional operations $(\wedge, \vee)$ and values $\{\text{true}, \text{false}\}$ of the Boolean semiring with the corresponding operations $(\times, \max)$ and values $[0, 1]$ of the Viterbi semiring. The study of various semiring interpretations of database query formalisms has a rich history motivated by the idea of *data provenance*.

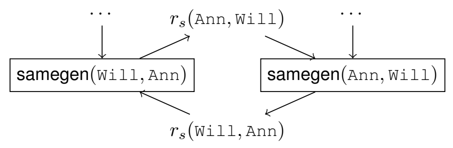


Figure 3: The rule r_s, "someone(x, y) :− samegen(y, x)", induces cycles in the clauses obtained at fixpoint. When unrolled into derivation trees such as those in Figure 2, these cycles result in the production of infinitely many derivation trees for a single output tuple.

The following result follows from Prop. 5.7 in [Green *et al.*, 2007], and concretizes the idea that DIFFLOG is a refinement of Datalog:

Theorem 3. *Let R be a set of candidate rules, and $\boldsymbol{w}$ be an assignment of weights $w_r \in [0, 1]$ to each of them, $r \in R$. Define $R_s = \{r \mid w_r \gneq 0\}$, and consider a potential output tuple t. Then, $v_t^{R,I}(\boldsymbol{w}) \gneq 0$ iff $t \in R_s(I)$.*

Furthermore, in the Appendix, we show that the output values $v_t(\boldsymbol{w})$ is well-behaved in its domain of definition:

Theorem 4. *The value of the output tuples, $v_t(\boldsymbol{w})$, varies monotonically with the rule weights $\boldsymbol{w}$, and is continuous in the region $0 < w_r < 1$.*

Note that DIFFLOG is discontinuous at boundary points when $w_r = 0$ or $w_r = 1$, and undefined outside the unit interval. To prevent this from causing problems during learning with gradient descent, we clamp the rule weights to the interval $[0.01, 0.99]$ in our implementation.

We could conceivably have chosen a different semiring in our definitions in Equations 1 and 2. One alternative would be to choose a space of events, corresponding to the inclusion of individual rules, and choosing the union and intersection of events as the semiring operations. This choice would make the system coincide with ProbLog [De Raedt *et al.*, 2007]. However, the #P-completeness of inference in probabilistic logics would make the learning process computationally expensive. Other possibilities, such as the arithmetic semiring $(\mathbb{R}, +, \times, 0, 1)$, would lead to unbounded values for output tuples in the presence of infinitely many derivation trees.

4.2 Evaluating and Automatically Differentiating DIFFLOG Programs

Because the set of derivation trees for an individual tuple t may be infinite, note that Equation 2 is merely *definitional*, and does not prescribe an algorithm to *compute* $v_t(\boldsymbol{w})$. Furthermore, numerical optimization requires the ability to automatically differentiate these values, i.e., to compute $\nabla_{\boldsymbol{w}} v_t$.

The key to automatic differentiation is tracking the *provenance* of each output tuple [Green *et al.*, 2007]. Pick an output tuple t, and let τ be its derivation tree with greatest value. For the purposes of this paper, the provenance of t is a map, $l_t = \{r \mapsto \#r \text{ in } \tau \mid r \in R\}$, which maps each rule r to the number of times it appears in τ. Given the provenance l_t of a tuple, observe that $v_t(\boldsymbol{w}) = \prod_{r \in R} w_r^{l_t(r)}$, so that the derivative of $v_t(\boldsymbol{w})$ can be readily computed as follows:

$$\frac{\partial v_t(\boldsymbol{w})}{\partial w_r} = \frac{l_t(r) v_t(\boldsymbol{w})}{w_r}. \tag{3}$$

Algorithm 1 EVALUATE$(R, \boldsymbol{w}, I)$, where R is a set of rules, $\boldsymbol{w}$ is an assignment of weight to each rule in R, and I is a set of input tuples.

```
1. Initialize the set of tuples in each relation, $F_P := \emptyset$, their valuations, $u(t) := 0$, and their provenance $l(t) = \{r \mapsto \infty \mid r \in R\}$.
2. For each input relation $P$, update $F_P := I_P$, and for each $t \in I_P$, update $u(t) := 1$ and $l(t) = \{r \mapsto 0 \mid r \in R\}$.
3. Until $(F, \boldsymbol{u})$ reach fixpoint,
   (a) Compute the immediate consequence of each rule, $r$, "$P_h(\boldsymbol{u}_h) :- P_1(\boldsymbol{u}_1), P_2(\boldsymbol{u}_2), \dots, P_k(\boldsymbol{u}_k)$":
       $$F'_{P_h} = \pi_{\boldsymbol{u}_h}(F_{P_1}(\boldsymbol{u}_1) \bowtie F_{P_2}(\boldsymbol{u}_2) \bowtie \cdots \bowtie F_{P_k}(\boldsymbol{u}_k)).$$
       Furthermore, for each tuple $t \in F'_{P_h}$, determine all sets of antecedent tuples, $A_g(t) = \{P_1(\boldsymbol{v}_1), P_2(\boldsymbol{v}_2), \dots, P_k(\boldsymbol{v}_k)\}$, which result in its production.
   (b) Update $F_{P_h} := F_{P_h} \cup F'_{P_h}$.
   (c) For each tuple $t \in F'_{P_h}$ and each $A_g(t)$: (i) compute $u'_t = w_r \prod_{i=1}^{k} u(P_i(\boldsymbol{v}_i))$, and (ii) if $u(t) < u'_t$, update:
       $$u(t) := u'_t, \text{ and } l(t) := \{r \mapsto 1\} + \sum_{i=1}^{k} l(P_i(\boldsymbol{v}_i)),$$
       where addition of provenance values corresponds to the element-wise sum.
4. Return $(F, \boldsymbol{u}, \boldsymbol{l})$.
```

In Algorithm 1, we present an algorithm to compute the output values $v_t(\boldsymbol{w})$ and provenance l_t, given R, $\boldsymbol{w}$, and the input tuples I. The algorithm is essentially an instrumented version of the "naive" Datalog evaluator [Abiteboul *et al.*, 1995]. We outline the proof of the following correctness and complexity claims in the Appendix.

Theorem 5. *Fix a set of input relations $\mathcal{I}$, output relations $\mathcal{O}$, and candidate rules R. Let* EVALUATE$(R, \boldsymbol{w}, I) = (F, \boldsymbol{u}, \boldsymbol{l})$. *Then: (a) $F = R(I)$, and (b) $\boldsymbol{u}(t) = v_t(\boldsymbol{w})$. Furthermore,* EVALUATE$(R, \boldsymbol{w}, I)$ *returns in time poly$(|I|)$.*

5 Formulating the Optimization Problem

We formulate the DIFFLOG synthesis problem as finding the value of the rule weights $\boldsymbol{w}$ which minimizes the difference between the output values of tuples, $v_t(\boldsymbol{w})$, and their expected values, 1 if $t \in O_+$, and 0 if $t \in O_-$. Specifically, we seek to minimize the L2 loss,

$$L(\boldsymbol{w}) = \sum_{t \in O_+} (1 - v_t(\boldsymbol{w}))^2 + \sum_{t \in O_-} v_t(\boldsymbol{w})^2. \qquad (4)$$

At the optimum point, Theorem 3 enables the recovery of a classical Datalog program from the optimum value $\boldsymbol{w}^*$.

Hybrid optimization procedure. In program synthesis, the goal is often to ensure exact compatibility with the provided positive and negative examples. We therefore seek zeros of the loss function $L(\boldsymbol{w})$, and solve for this using Newton's root-finding algorithm: $\boldsymbol{w}^{(i+1)} := \boldsymbol{w}^{(i)} - L(\boldsymbol{w})\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) / \|\nabla_{\boldsymbol{w}} L(\boldsymbol{w})\|^2$. To escape from local minima and points of slow convergence, we periodically intersperse iterations of the MCMC sampling, specifically simulated annealing.

Forbidden rules. If a single rule $r \in R$ is seen to independently derive an undesirable tuple $t \in O_-$, i.e., if $l_t(r) \geq 1$ and $l_t(r') = 0$ for all $r \neq r$, then it is marked as a *forbidden* rule, and its weight is immediately clamped to 0: $w_r^{(i+1)} := 0$.

Learning details. We initialize $\boldsymbol{w}$ by uniformly sampling weights $w_r \in [0.25, 0.75]$. We apply MCMC sampling after every 30 iterations of Newton's root-finding method, and sample new weights as follows:

$$X \sim U(0,1)$$
$$w_{new} = \begin{cases} w_{old}\sqrt{2X} & \text{if } X < 0.5 \\ 1 - (1 - w_{old})\sqrt{2(1-X)} & \text{otherwise.} \end{cases}$$

The temperature T used in simulated annealing is as follows:

$$T = \frac{1.0}{C * log(5 + \#iter)}$$

where C is initially 0.0001 and $\#iter$ is the number of iterations. We accept the newly proposed sample with probability

$$p_{acc} = \min(1, \pi_{new}/\pi_{curr}),$$

where $\pi_{curr} = \exp(-L_2(\boldsymbol{w}_{curr})/T)$ and $\pi_{new} = \exp(-L_2(\boldsymbol{w}_{new})/T)$.

Separation-guided search termination. After computing each subsequent $\boldsymbol{w}^{(i)}$, we examine the provenance values for each output tuple to determine whether the current position can directly lead to a solution to the rule selection problem. In particular, we compute the sets of desirable—$R_+ = \{r \in l(t) \mid t \in O_+\}$—and undesirable rules—$R_- = \{r \in l(t) \mid t \in O_-\}$, and check whether $R_+ \cap R_- = \emptyset$. If these sets are separate, then we examine the candidate solution R_+, and return if it satisfies the output specification.

6 Empirical Evaluation

Our experiments address the following aspects of DIFFLOG:

1. Its effectiveness at synthesizing Datalog programs and comparison to the state-of-the-art tool ALPS [Si *et al.*, 2018], which already outperforms existing ILP tools [Albarghouthi *et al.*, 2017; Muggleton *et al.*, 2015] and supports relations with arbitrary arity, sophisticated joins, and predicate invention;
2. the benefit of employing MCMC search compared to a purely gradient-based method; and
3. scaling with number of training labels and rule templates.

We evaluated DIFFLOG on a suite of 34 benchmark problems [Si *et al.*, 2018]. This collection draws benchmarks from three different application domains: (*a*) knowledge discovery, (*b*) program analysis, and (*c*) relational queries. The characteristics of the benchmarks are shown in the Appendix. These benchmarks involve up to 10 target rules, which could be recursive and involve relations with arity up to 6. The implementation of DIFFLOG comprises 4K lines of Scala code. We use Newton's root-finding method for continuous optimization and apply MCMC-based random sampling every 30 iterations. All experiments were conducted on Linux machines with Intel Xeon 3GHz processors and 64GB memory.

Benchmark	Rel	Rule		Tuple		DIFFLOG			ALPS
		Exp	Cnd	In	Out	Iter	Smpl	Time	Time
`inflamation`	7	2	134	640	49	1	0	**1**	2
`abduce`	4	3	80	12	20	1	0	< **1**	2
`animals`	13	4	336	50	64	1	0	**1**	40
`ancestor`	4	4	80	8	27	1	0	< **1**	14
`buildWall`	5	4	472	30	4	5	1	**7**	67
`samegen`	3	3	188	7	22	1	0	**2**	12
`scc`	3	3	384	9	68	6	1	**28**	56
`polysite`	6	3	552	97	27	17	1	**27**	84
`downcast`	9	4	1,267	89	175	5	1	**30**	1,646
`rv-check`	5	5	335	74	2	1,205	41	**22**	195
`andersen`	5	4	175	7	7	1	0	**4**	27
`1-call-site`	9	4	173	28	16	4	1	**4**	106
`2-call-site`	9	4	122	30	15	25	1	**53**	676
`1-object`	11	4	46	40	13	3	1	**3**	345
`1-type`	12	4	70	48	22	3	1	**4**	13
`escape`	10	6	140	13	19	2	1	**1**	5
`modref`	13	10	129	18	34	1	0	**1**	2,836
`sql-10`	3	2	734	10	2	7	1	**11**	41
`sql-14`	4	3	23	11	6	1	0	< **1**	54
`sql-15`	4	2	186	50	7	902	31	875	**11**

Table 1: Characteristics of benchmarks and performance of DIFFLOG compared to ALPS. Rel shows the number of relations. The columns titled Rule represent the number of expected and candidate rules. Tuple shows the number of input and output tuples. Iter and Smpl report the number of iterations and MCMC samplings. Time shows the running time of DIFFLOG and ALPS in seconds.

6.1 Effectiveness

We first evaluate the effectiveness of DIFFLOG and compare it with ALPS. The running time and solution of DIFFLOG depends on the random choice of initial weights. DIFFLOG exploits this characteristic by running multiple synthesis processes for each problem in parallel. The solution is returned once any one of the parallel processes successfully synthesizes a Datalog program which is consistent with the specifications. We populated 32 processes in parallel and measured the running time until the first solution was found. The timeout is set to 1 hour for each problem.

Table 1 shows the running of DIFFLOG and ALPS. Of the 34 benchmarks, we excluded 14 benchmarks where either both DIFFLOG and ALPS find solutions within a second (13 benchmarks) or both solvers time-out (1 benchmark). DIFFLOG outperforms ALPS on 19 of the remaining 20 benchmarks in Table 1. In particular, DIFFLOG is orders of magnitude faster than ALPS on most of the program analysis benchmarks. Meanwhile, the continuous optimization may not be efficient when the problem has many local minimas and the space is not convex. For example, `sql-15` has a lot of sub-optimal solutions that generate not only all positive output tuples but also some negative ones.

Figure 4 depicts the distribution of running time on the benchmarks. The results show that DIFFLOG is always able to find solutions for all the benchmarks except for occasional timeouts on `downcast`, `rv-check`, `scc`, and `sql-15`. Also note that even the median running time of DIFFLOG is smaller than the running time of ALPS for 13 out of 20 benchmarks.

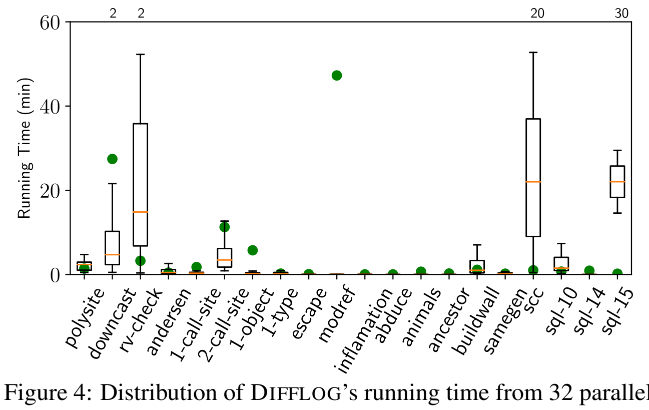


Figure 4: Distribution of DIFFLOG's running time from 32 parallel runs. The numbers on top represents the number of timeouts. Green circles represent the running time of ALPS.

6.2 Impact of MCMC-based Sampling

Next, we evaluate the impact of our MCMC-based sampling by comparing the performance of three variants of DIFFLOG: (*a*) a version that uses both Newton's method and the MCMC-based technique (**Hybrid**), which is the same as in Section 6.1, (*b*) a version that uses only Newton's method (**Newton**), and (*c*) a version that uses only the MCMC-based technique (**MCMC**). Table 2 shows the running time of the best run and the number of timeouts among 32 parallel runs for these three variants. The table shows that our hybrid approach strikes a good balance between exploitation and exploration. In many cases, **Newton** gets stuck in local minima; for example, it cannot find any solution for `rv-check` within one hour. **MCMC** cannot find any solution for 6 out of 10 benchmarks. Overall, **Hybrid** outperforms both **Newton** and **MCMC** by reporting 31× and 54× fewer timeouts, respectively.

6.3 Scalability

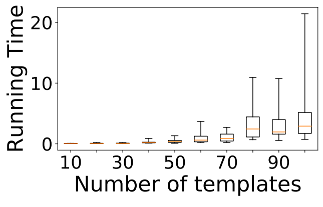
Finally, we evaluate the scalability of DIFFLOG-based synthesis, which is affected by two factors: the number of templates and the size of training data. Our general observation is that increasing either of these does not significantly increase the effective running time (i.e., the best of 32 parallel runs).

Figure 5 shows how running time increases with the number of templates.[1] As shown in Figure 5a, the running time distribution for `2-call-site` tends to have larger variance when the number of templates increases, but the best running time (out of 32 i.i.d samples) only increases modestly. The running time distribution for `downcast`, shown in Figure 5b, has a similar trend except that smaller number of templates does not always lead to smaller variance or faster running time. For instance, the distribution in the setting with 180 templates has larger variance and median than distributions in the subsequent settings with larger number of templates. This indicates that the actual combination of templates also matters. In general, approximately half the benchmarks follow a trend similar to Figure 5a, with monotonically increasing variance in running times, while the remaining benchmarks are similar to Figure 5b.

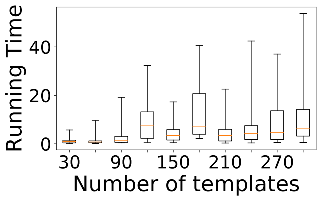
[1]We ensure that all candidate rules in a set are also present in subsequent larger sets.

Benchmark	Hybrid			Newton			MCMC		
	Best	Median	Timeout	Best	Median	Timeout	Best	Median	Timeout
`polysite`	27s	142s	0	10s	72s	0	12s	76s	0
`downcast`	30s	310s	2	16s	252s	9	70s	268s	7
`rv-check`	22s	948s	2	N/A	N/A	32	N/A	N/A	32
`andersen`	4s	29s	0	3s	15s	10	4s	17s	9
`1-call-site`	4s	18s	0	8s	18s	1	N/A	N/A	32
`2-call-site`	53s	225s	0	27s	N/A	17	42s	94s	9
`1-object`	3s	17s	0	3s	N/A	17	N/A	N/A	32
`1-type`	4s	12s	0	3s	N/A	18	N/A	N/A	32
`escape`	1s	2s	0	1s	N/A	17	N/A	N/A	32
`modref`	1s	2s	0	1s	1s	4	N/A	N/A	32
Total			4			125			217

Table 2: Effectiveness of MCMC sampling in terms of the best and median running times and the number of timeouts observed over 32 independent runs.

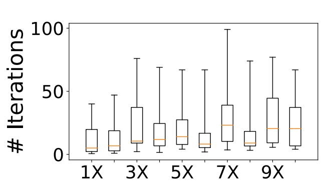


(a) `2-call-site`

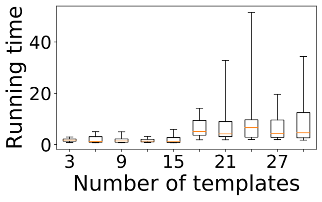


(b) `downcast`

Figure 5: Running time distributions (in minutes) for `downcast` and `2-call-site` with different number of templates.



(a)



(b)

Figure 6: Performance of DIFFLOG on `andersen` with different sizes of data: (a) the distribution of number of iterations, (b) the distribution of running time (in seconds).

The size of training data is another important factor affecting the performance of DIFFLOG. Figure 6a shows the distribution of the number of iterations for `andersen` with different sizes of training data. According to the results, the size of training data does not necessarily affect the number of iterations of DIFFLOG. Meanwhile, Figure 6b shows that the end-to-end running time increases with more training data. This is mainly because more training data imposes more cost on the DIFFLOG evaluator. However, the statistics show that the running time increases linearly with the size of data.

7 Conclusion

We have presented a technique to synthesize Datalog programs using numerical optimization. The central idea is to formulate the problem as an instance of rule selection, and then relax classical Datalog to a refinement named DIFFLOG. In a comprehensive set of experiments, we show that by learning a DIFFLOG program and then recovering a classical Datalog program, we can achieve significant speedups over the state-of-the-art Datalog synthesis systems. In future, we plan to extend the approach to other synthesis problems such as SyGuS and to applications in differentiable programming.

Acknowledgments

We thank the reviewers for their insightful feedback. This research was supported in part by NSF awards #1737858 and #1836936, and by a Facebook Research Award.

References

[Abiteboul *et al.*, 1995] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Albarghouthi *et al.*, 2017] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of Datalog programs. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming*, CP, 2017.

[Alur *et al.*, 2015] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

[Bravenboer and Smaragdakis, 2009] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2009.

[Cohen and Page, 1995] William W. Cohen and C. David Page. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Comput.*, 13(3&4):369–409, 1995.

[De Raedt *et al.*, 2007] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th*

International Joint Conference on Artificial Intelligence, 2007.

[Embar *et al.*, 2018] Varun Embar, Dhanya Sridhar, Golnoosh Farnadi, and Lise Getoor. Scalable structure learning for Probabilistic Soft Logic. *CoRR*, abs/1807.00973, 2018.

[Evans and Grefenstette, 2018] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data (Extended abstract). In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, 2018.

[Green *et al.*, 2007] Todd Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th Symposium on Principles of Database Systems*, PODS, 2007.

[Gulwani, 2011] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Symposium on Principles of Programming Languages*, POPL, 2011.

[Kimmig *et al.*, 2011] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. An algebraic Prolog for reasoning about possible worlds. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[Kimmig *et al.*, 2017] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *J. of Applied Logic*, 22(C), July 2017.

[Kok and Domingos, 2005] Stanley Kok and Pedro M. Domingos. Learning the structure of Markov Logic Networks. In *Machine Learning, Proceedings of the Twenty-Second International Conference*, 2005.

[Liang *et al.*, 2010] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[Loo *et al.*, 2006] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David Gay, Joseph Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *Proceedings of the 2006 International Conference on Management of Data*, SIGMOD, 2006.

[Manhaeve *et al.*, 2018] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*, 2018.

[Muggleton *et al.*, 2015] Stephen Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited. *Machine Learning*, 100(1), 2015.

[Muggleton, 1995] Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3&4), 1995.

[Muggleton, 1996] Stephen Muggleton. Stochastic logic programs. In *New Generation Computing*. Academic Press, 1996.

[Poole, 1995] David Poole. Logic programming for robot control. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, IJCAI, 1995.

[Richardson and Domingos, 2006] Matthew Richardson and Pedro Domingos. Markov Logic Networks. *Machine Learning*, 62(1-2), 2006.

[Rocktäschel and Riedel, 2017] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, 2017.

[Schkufza *et al.*, 2016] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Commun. ACM*, 59(2), 2016.

[Seo, 2018] Jiwon Seo. Datalog extensions for bioinformatic data analysis. In *40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, EMBC, 2018.

[Shapiro, 1983] Ehud Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

[Shkapsky *et al.*, 2016] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with Datalog queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD, 2016.

[Si *et al.*, 2018] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of Datalog programs. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, FSE, 2018.

[Wang *et al.*, 2014] William Yang Wang, Kathryn Mazaitis, and William Cohen. Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, 2014.

[Yang *et al.*, 2017] Fan Yang, Zhilin Yang, and William W. Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems*, 2017.