Learning Programs as Logical Queries

Type 1 Paper

Charles Jordan[1⋆] and Łukasz Kaiser[2]

[1] ERATO Minato Project, JST & Hokkaido University
[2] LIAFA, CNRS & Université Paris Diderot
skip@ist.hokudai.ac.jp, kaiser@liafa.univ-paris-diderot.fr

Abstract. Program learning focuses on the automatic generation of programs satisfying the goal of a teacher. One common approach is counter-example guided inductive synthesis, where we generate a sequence of candidate programs and the teacher responds with counter-examples for which the candidate fails. In this paper we focus on a logical approach, where programs are tuples of logical formulas, i.e. logical queries, and inputs and outputs are relational structures. We introduce our model of inductive synthesis and our implementation of it using SAT and QBF solvers. We survey basic theoretical properties of our model and show a few experimental results: learning complexity-theoretic reductions, polynomial-time programs, and learning board games from examples.

1 Introduction

The dream of program learning – that instead of writing code we will teach computers to derive it automatically – has a long history in computer science and has appeared in many forms. In program synthesis we seek any program that satisfies a specification, usually given as a logical formula. However, writing this formal logical specification can be as hard as writing the program itself. To avoid formal specifications, one can consider learning programs using only a set of example inputs and outputs.

Program learning is difficult and there are many theoretical and practical challenges. Specifying the exact requirements for the learner is already hard. One must choose a representation of programs, what assumptions on efficiency to make, whether the goal is any program or a short one, and the precise meaning of "short". More practically, searching for programs in any standard programming language is very inefficient. Most problems are undecidable, there are hardly any useful normal forms to exploit, and modifying even a small part of a program can dramatically change its behavior. Given these issues, it is easy to understand why the dream of program learning has not yet been realized.

Choosing a representation for programs is critical and recent work [1,8,9] in descriptive complexity suggests that *logical queries* are uniquely suited for this task. Descriptive complexity studies the connection between logics and complexity

⋆ This work was supported in part by KAKENHI Grant Number 25106501.

classes: logics equivalent to classes such as NL, P, NP, and PSPACE are known. Searching for formulas in these logics corresponds to searching for programs in these classes, and has other advantages that we discuss below.

Here, we introduce a model of learning programs represented by logical queries. Our approach tries to address the theoretical challenges with program learning mentioned above, and we leverage recent advances in SAT and QBF solvers to address more practical concerns. We have implemented our model[3] and we describe three kinds of initial experiments. For program synthesis, we describe learning complexity-theoretic reductions and also polynomial-time equivalents for NP programs. Both of these tasks have complete specifications. To show that this is not a limitation of our approach, we experiment with learning the rules of board games from sets of example plays. While these experiments are preliminary, we believe the results show that our approach to program learning is promising.

Related Work. Much of our motivation comes from recent papers using ideas from descriptive complexity in inductive synthesis. For example, given a specification in an expressive logic (second-order), [8] synthesized equivalent formulas in less expressive logics which can be evaluated more efficiently. Automatically finding complexity-theoretic reductions between computational problems was first considered by [1]. They focused on quantifier-free reductions, a weak class of reduction defined by tuples of quantifier-free formulas.

Both problems are essentially the same – finding formulas in a particular form that satisfy desired properties. However, the implementations are separate and not publicly available. We [9] have previously compared a number of different approaches to reduction finding. In this paper, we introduce a more general approach – allowing the user to specify an "outline" of the desired formula and a specification that it must satisfy. We provide a single model and freely available implementation that can be used to experiment with various synthesis problems.

Inductive synthesis has a long history and there is a tremendous amount of work that we do not cover here. See, e.g., [4,11] for a more general perspective.

2 Background in Logic and Descriptive Complexity

In this section we briefly review the necessary background from descriptive complexity. For more details, see [7] or Chapter 3 of [3].

There are many possible representations of programs; in this paper, we focus on *logical* representations. One benefit of the logical approach is that we are able to treat structures such as graphs directly, instead of encoding them into words or numbers. This allows us to express many interesting programs succintly, and formulas have natural normal forms. These provide guidance for hypothesis spaces, and improve understandability of learned programs.

Fundamentally, programs transform given inputs into outputs; we represent these inputs and outputs as relational structures (for example, graphs or binary strings). A *relational signature* is a tuple of predicate symbols R_i with arities a_i

[3] All source code is freely available in the 0.9 release of Toss at `http://toss.sf.net`.

and constant symbols c_j, $\tau := (R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s)$. A finite *$\tau$-structure* consists of a finite universe U, an a_i-ary relation for each predicate symbol of τ, and a definition – an element of U – for each constant symbol:

$$A := (U, R_1 \subseteq U^{a_1}, \ldots, R_r \subseteq U^{a_r}, c_1 \in U, \ldots, c_s \in U)\,.$$

For example, the signature for directed graphs contains a single, binary predicate symbol E and so a directed graph consists of a finite set of vertices and a binary edge relation. We denote the set of all finite τ-structures by $\mathsf{Struc}(\tau)$. Our programs are built from formulas in various logics. Formulas of *first-order logic* over a signature τ have the form

$$\varphi \;:=\; R_i(x_1, \ldots, x_{a_i}) \mid x_i = x_j \mid x_i = c_j \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x_i\, \varphi \mid \forall x_i\, \varphi,$$

where $x_1, x_2, \ldots$ are first-order variables, and the semantics, given an assignment of the variables x_i to elements e_i of the structure, is defined in the natural way.

Queries. Single formulas can be used to define properties or decision problems, but in general we represent programs as queries. Queries map σ-structures to τ-structures, defining the universe, relations, and constants using logical formulas. A first-order query from σ-structures to τ-structures is an $r + s + 2$-tuple,

$$q \;:=\; (k, \varphi_0, \varphi_1, \ldots, \varphi_r, \psi_1, \ldots, \psi_s)\,.$$

The number $k \in \mathbb{N}$ is the *dimension* of the query. Each φ_i, ψ_j is a first-order formula over the signature σ. Let A be a σ-structure with universe U^A. The formula φ_0 has free variables $x_1, \ldots, x_k$ and defines the universe U of $q(A)$,

$$U \;:=\; \big\{(u_1, \ldots, u_k) \mid u_i \in U^A, A \models \varphi_0(u_1, \ldots, u_k)\big\}\,.$$

That is, the new universe consists of k-tuples of elements of the old universe, where φ_0 determines which k-tuples are included.

Each remaining φ_i has free variables $x_1^1, \ldots, x_1^k, x_2^1, \ldots, x_{a_i}^k$ and defines

$$R_i := \big\{(u_1^1, \ldots, u_1^k), \ldots, (u_{a_i}^1, \ldots, u_{a_i}^k) \mid A \models \varphi_i(u_1^1, \ldots, u_{a_i}^k)\big\} \cap U^{a_i}\,.$$

That is, φ_i determines which of the a_i-tuples of U are included in R_i. Finally, each ψ_i has free variables $x_1, \ldots, x_k$ and defines c_i as the unique $(u_1, \ldots, u_k) \in U$ such that $A \models \psi_i(u_1, \ldots, u_k)$.

First-order queries therefore transform σ-structures into τ-structures, and we write $q(A)$ to represent the resulting τ-structure. The restriction to first-order logic here is not essential – given a logic $\mathcal{L}$, we define $\mathcal{L}$-queries in a similar way.

Extending first-order logic. So far, we have focused only on first-order logic. However, first-order logic on finite structures is often too limited from the computational perspective – it cannot express many interesting queries that are easy to compute. Over finite structures with additional numeric predicates, the first-order definable properties correspond exactly to uniform AC^0 (cf. [7]).

To remove this limitation, one extends first-order logic with various operators. For example, the *transitive closure operator* allows us to write formulas of the

form $\mathrm{TC}[x_1, x_2.\varphi(x_1, x_2)](y_1, y_2)$. This formula takes the transitive and reflexive closure of the (implicit) relation defined by $\varphi(x_1, x_2)$ and evaluates it on (y_1, y_2). The *least fixed-point operator* allows recursive definitions in formulas of the form $\mathrm{LFP}[R(x_1, \ldots, x_k) = \varphi(R, x_1, \ldots, x_k)](y_1, \ldots, y_k)$, where R is a new relation symbol appearing only positively in the inner formula φ. The result of this operator is defined as the least fixed-point of the operator $R(\overline{x}) \to \varphi(R, \overline{x})$. Finally, *second-order logic* (SO) allows to use quantifiers over relations.

There are many known correspondences between logics and complexity classses (see [7]). The oldest result [2] shows that the class NP is captured by existential second-order logic. This implies that coNP is captured by universal second-order logic, and that full SO captures the polynomial-time hierarchy. More practically, polynomial-time computations are captured by least fixed-point logic (LFP) when a linear order relation is present [5,12]. Although LFP is presumably more expressive than transitive closure logic (TC), TC captures all problems solvable in non-deterministic logarithmic space (NL) on ordered structures [6].

Outlines. For a logic $\mathcal{L}$, we refer to the set of $\mathcal{L}$-formulas in which atoms a may be guarded by some Boolean guard[4] G_a as *$\mathcal{L}$-formula outlines*. The Boolean guards are intended to mean "a occurs here", and given an *instantiation* of the guards I, we can instantiate an $\mathcal{L}$-formula outline ψ to an $\mathcal{L}$-formula ψ^I by replacing each $G_a a$ by a if G_a is true in I, and leaving the atom empty otherwise. Intuitively, an outline fixes the structure of the formula but not the precise contents. We refer to queries containing $\mathcal{L}$-formula outlines as *$\mathcal{L}$-query outlines*. We omit $\mathcal{L}$ when it is clear from context, and use *outline* to refer to both query and formula outlines. Given an outline o, we write $\mathsf{inst}(o)$ for the set of formulas or queries obtainable as instantiations of o.

3 Learning Logical Queries

In this section, we introduce our model of learning logical queries. The model consists of a learner giving candidate queries or hypotheses and a teacher (or verifier), which gives counter-examples or accepts the query. A learning task is characterized by a few parameters, first is the target class $\mathcal{C}$.

Let $\mathcal{C} \subseteq \mathsf{Struc}(\tau) \times \mathsf{Struc}(\sigma)$ be a binary relation on relational structures, and define the domain of $\mathcal{C}$ as $\mathrm{dom}(\mathcal{C}) = \{A \mid (A, B) \in \mathcal{C} \text{ for some } B\}$.

Definition 1. *A $\mathcal{C}$-*teacher *t is a function*

$$t\colon (\tau \to \sigma)\text{-}queries \to (dom(\mathcal{C}) \times SO(\sigma)) \cup \{\bot\}$$

that satisfies the following condition.

$$t(q) = \begin{cases} \bot & \text{if } \{(A, q(A)) \mid A \in dom(\mathcal{C})\} \subseteq \mathcal{C}, \\ (A, \psi) & \text{where } A \in dom(\mathcal{C}), (A, q(A)) \notin \mathcal{C}, (B \models \psi \text{ iff } (A, B) \in \mathcal{C}). \end{cases}$$

[4] We do not require that identical atoms share guards, that distinct atoms have different guards, or that all atoms are guarded.

That is, a teacher accepts a query q if for all $A \in \mathrm{dom}(\mathcal{C})$ we have $(A, q(A)) \in \mathcal{C}$, and otherwise replies with a counter-example (A, ψ) such that $q(A) \not\models \psi$. Here, the formula ψ defines the acceptable output on A. Of course, in practice we generally restrict attention to computable teachers and "reasonable" classes $\mathcal{C}$.

Definition 2. *Let $\mathcal{H}$ be a class of logical queries. An $\mathcal{H}$-*learner *L is a function that, given a sequence of examples $((A_1, \psi_1), \ldots, (A_m, \psi_m))$, satisfies*

$$L((A_1, \psi_1), \ldots, (A_m, \psi_m)) = \begin{cases} h, & h \in \mathcal{H}, h(A_i) \models \psi_i \text{ for } 1 \leq i \leq m; \\ \bot, & \text{if no such } h \in \mathcal{H} \text{ exists}. \end{cases}$$

Note that our learners must always be consistent, and they return $\bot$ iff there is no consistent query in the hypothesis space. The logic used in the query is determined by $\mathcal{H}$. In practice, we always use *outline learners*, which are defined as $\mathcal{H}$-learners for $\mathcal{H} = \mathsf{inst}(q)$ for some query outline q.

The outline gives a compact representation of a hypothesis space, and can also enforce certain restrictions on the query. For example, outlines can require a query to generate an *extension*[5] of the structure, which is useful when searching for programs to give explicit isomorphisms or satisfying solutions to SAT instances.

Outline learners using QBF solvers. Essentially, an outline learner is a learner with a finite, uniform hypothesis space. The main condition of an outline learner, namely that $\bigwedge_i q^I(A_i) \models \psi_i$, can be translated into a QBF formula. This is done by introducing a Boolean variable for each tuple in each relation in $q^I(A_i)$, setting it to true only if the respective formula from q holds for the specified tuple, and then translating ψ_i using these Boolean variables for relations. We omit the complete algorithm due to space constraints, but it is crucial that all logics we consider are expressible in SO and that each $\psi_i \in \mathrm{SO}$, by the definition of a teacher. This allows us to implement outline learners using QBF solvers to search for a suitable instantiation of the guards. For a query outline q and a QBF solver S, we write $L(S, q)$ for the $\mathsf{inst}(q)$-learner using S as explained above.

Given an $\mathcal{H}$-learner L and a $\mathcal{C}$-teacher t we define the sequence L_i^t of the interactions between L and t inductively as follows. We set $L_0^t := L()$, the hypothesis that L returns on an empty list of examples. If for some i we get $L_i^t = \bot$ then the sequence is finished – there is no $h \in \mathcal{H}$ that satisfies the teacher. Else, let $E_i := t(L_i^t)$ be the answer of the teacher to L_i^t. If $E_i = \bot$ the sequence L_i^t is finished, the last hypothesis was accepted. In the other case, set $L_{i+1}^t = L(E_0, \ldots, E_i)$. The following properties are immediate.

Theorem 3. *Let S be a correct QBF solver, t a $\mathcal{C}$-teacher, and q a query outline.*

1. *$L(S, q)$ is a consistent and conservative $\mathsf{inst}(q)$-learner.*
2. *If $t(h) = \bot$ for some $h \in \mathsf{inst}(q)$ then the sequence $L(S, q)_i^t$ is finite and its last element g satisfies $t(g) = \bot$.*
3. *If there is no $h \in \mathsf{inst}(q)$ for which $t(h) = \bot$ then the sequence $L(S, q)_i^t$ is finite and its last element is $\bot$.*

[5] An *extension* of a structure is formed by adding new predicates while leaving existing predicates unchanged.

Of course, we often have to restrict the teacher in order to obtain decidability. We usually restrict $\mathcal{C}$ to be a finite set, and usually implement the teacher using SAT or QBF solvers as well[6]. Although one can easily construct instances where arbitrarily-large examples are required, in practice it seems that queries that are correct on moderately sized examples are almost always correct in general (where "moderate" depends on the complexity of the query).

A *learning task* is specified by the pair $(\mathcal{C}, \mathcal{H})$. We write $L(t, q)$ for the last element of the sequence $L(S, q)_i^t$, where S is a chosen solver. The sequence is always finite by Theorem 3, so $L(t, q)$ is well-defined. In the next section, we examine various learning tasks and look for reasonable teachers and outlines.

4 Experimental Results

4.1 Learning Reductions

Learning reductions was first considered by [1], and we have also [9] implemented, benchmarked and evaluated a number of different approaches to the problem.

Problem. In descriptive complexity, a reduction from the τ-property defined by φ to the σ-property defined by ψ is a $(\tau \to \sigma)$-query q that satisfies

$$A \models \varphi \iff q(A) \models \psi \tag{1}$$

for all τ-structures A. Of course, reductions should have less computational power than the complexity classes they are used in and descriptive complexity focuses on first-order reductions (i.e., first-order queries as reductions). Here, we focus on quantifier-free first-order reductions, a weaker class that still suffices to capture important complexity classes (cf. [9] for more details).

In order to make finding quantifier-free reductions decidable, we restrict attention to a fixed size n, i.e., we require Formula (1) to hold only for structures of size at most n. Assume that we are searching for a dimension-k reduction from the τ-property defined by φ to the σ-property defined by ψ.

Let P be the set of τ-structures of size at most n that satisfy φ, Q be the set of σ-structures of size at most n^k that satisfy ψ, and $\overline{P}$ and $\overline{Q}$ be their complements up to the size bounds. Our target class is $\mathcal{C} = (P \times Q) \cup (\overline{P} \times \overline{Q})$ – we want a query that maps positive instances to positive instances and negative instances to negative instances.

Outline. As an outline, we focus on reductions in which all formulas are in DNF with c conjunctions. We fix φ_0 to be always true and the dimension k (so the new universe is the set of k-tuples of elements of the old universe). Finally, we have a number of parameters determining the atomic formulas that may occur – for example, whether to allow certain numeric predicates such as successor.

[6] We recommend GlueMiniSat as a SAT and RAReQS as a QBF solver, c.f. [9].

Teacher. When the teacher receives a candidate hypothesis q, it checks Formula (1), i.e. $A \models \varphi \iff q(A) \models \psi$ for all structures A of size at most n. This is done by assigning a Boolean variable to each bit of A and $q(A)$, and then constructing a QBF (or SAT) instance similar to the learner $L(S, q)$ described above. If the instance is satisfied, the assignment returned by the solver is used to construct a structure A that is a counter-example to the hypothesis q. The teacher returns (A, ψ) if $A \models \varphi$ and $(A, \neg\psi)$ otherwise.

Results. We refer to [9] for more details on our reduction-finding experiments. Our approach using GlueMiniSat as solver significantly improves upon the previous, specialized program ReductionFinder [1].

Example. Consider the NL-complete problems of reachability (given a directed graph with labeled vertices s and t, determine whether t is reachable from s) and all-pairs reachability (determine if a directed graph is strongly connected).

$$\text{Reach} := TC[x, y.E(x, y)](s, t) \quad \text{AllReach} := \forall x_1, x_2\ (TC[y, z.E(y, z)](x_1, x_2))\,.$$

Our system finds the following correct reduction for $n \geq 3$ in less than a second:

$$(k := 1,\ \varphi_0 := \mathit{true},\ \varphi_1 := x_1 = s \vee x_2 = t \vee E(x_2, x_1))\ .$$

This reverses all edges in the original graph, adds directed edges from s to all vertices and also adds directed edges to t from all vertices. A similar reduction exists without reversing the edges – however the above is our actual output.

4.2 Learning Fast Programs

In this section, we consider synthesizing programs for a given logical specification. This is similar to [8], however they focused on synthesizing formulas in more specialized logics.

Problem. In program synthesis, we are given a specification and hope to find an efficient program satisfying it. For us, a specification is a way to verify whether the output $q(A)$ is accepted for A. There are two major variations – either the output for each structure is unique (as in our example here), or there is a set of acceptable outputs (e.g., when finding a satisfying solution for SAT instances).

In our example here, we have a query s in an expressive logic (SO) and wish to find an equivalent query in a less-expressive logic. In particular, we consider the problem of identifying winning regions in finite games – i.e., directed graphs with a predicate $V_0(x)$ meaning that vertex x belongs to Player 0. Decidability requires restricting the size of the games n, and we set $\mathcal{C}$ to be the set of pairs (A, B) such that A is a finite game of size at most n and B is the extension of A with the winning region identified in a new monadic predicate W.

Outline. Similar to the case for reductions, we use several parameters to fix an outline. For reductions, we considered only quantifier-free formulas – however, in this section we need more expressive power. We focus on least fixed-point formulas, with a single fixed-point operator that is outermost[7]. We fix the arity a of the fixed-point predicate, and assume that the inner formula is a disjunction of l quantified CNF formulas with k variables each. We use such an outline for exactly one selected relation W in the query, all others are set to identity.

Teacher. Assume that we have a SO query s that produces the desired extension with the winning region identified. Given a hypothesis q, the teacher can again use a SAT or QBF solver (as was the case for reductions) to guess a game A of size at most n such that in $q(A)$ the new relation $W(x)$ is not equivalent to the region in $s(A)$. Let $a_1, \ldots, a_k$ be the winning positions in $s(A)$. The teacher then returns the pair $\big(A, \forall x\ (W(x) \leftrightarrow \bigvee_i (x = a_i))\big)$.

Example. Consider the case of identifying winning regions in finite reachability games. When the current vertex belongs to a player, that player chooses an outgoing edge and moves to a connected vertex. Player 1 loses if the play reaches a vertex that belongs to her and has no outgoing edge. Similarly, Player 0 loses if the play reaches a position where he must but cannot move, but also if the play becomes a cycle and goes on forever. The goal is to identify the vertices from which Player 0 has a winning strategy. That is, we want a formula $\varphi(x)$ which holds exactly on the vertices for which Player 0 has a winning strategy.

Reachability games are *positional*, i.e., it suffices to consider strategies that depend only on the current position and not on the history of the game. Therefore, a strategy of Player 1 can be defined as a binary relation S_1 that is a subset of the edges and that, for vertices of Player 0, contains all successors of the vertex as well. Then, x is winning for Player 1 (by S_1) if all vertices reachable from x by S_1 either belong to Player 0 or have an S_1-successor. This is easily expressible using the TC operator and guessing the strategy leads to a second-order formula $\varphi_0(x)$ which holds exactly if x is winning for Player 0.

In reachability games, the following LFP formula defines the winning region for Player 0:

$$\mathrm{LFP}\Big[W(x) = \Big\{ (\neg\exists x_1 : (\neg W(x_1) \wedge E(x, x_1)) \wedge \neg V_0(x)) \quad \vee \\ \exists x_1 : (W(x_1) \wedge E(x, x_1) \wedge V_0(x)) \Big\}\Big](x)\,.$$

The LFP operator recursively defines $W(x)$, starting with the empty set and adding tuples that satisfy the formula on the right until a fixed-point is reached. Therefore, this formula says that a position x is winning for Player 0 if
(a) it is the opponent's move and all outgoing edges go to positions we win; or
(b) it is Player 0's move and there is an edge to a winning position.

Recall that the fixed-point predicate is initially empty. Therefore, the winning positions after one iteration are the positions belonging to the opponent with no

[7] This is a normal-form for fixed-point logics, although the arity of the fixed point may increase if it is required to be outermost.

outgoing edges. Then, the winning region grows gradually until it is the *attractor* of those positions – which is correct. An equivalent, slightly longer formula is found by our program in less than a minute for $n \geq 3$.

Further work. The LFP formula for reachability games can be written by hand, but our motivation for presenting this example is the hope to compute polynomial-time solvers for other games. In particular, weak parity games and parity games are also positional, so it is trivial to write a SO formula defining the winning region (as we did for reachability games). But the polynomial-time program for solving weak parity games is complicated, and the existence of a polynomial-time solver for full parity games (which is equivalent to the existence of an LFP formula) is a long-standing open problem.

Our implementation can also search for other programs, for example, programs for graph isomorphism, graph coloring or SAT. There are natural classes where these problems are in polynomial-time[8].

4.3 Learning Formulas from Examples

Recently, a system was implemented [10] that represents board games as relational structures and learns their rules from observing example play videos. Fundamentally, the system works by computing minimal distinguishing formulas for sets of structures, e.g., formulas satisfied by structures representing winning positions and by none of the losing ones. We implement the computation of distinguishing formulas in our framework and compare the performance.

Problem. Let $\mathcal{P}$ and $\mathcal{N}$ be finite sets of τ-structures. We want to learn a formula φ without free variables such that $A \models \varphi$ for all $A \in \mathcal{P}$ and for no $A \in \mathcal{N}$. Unlike previous tasks, we want a minimal such formula, not just an arbitrary one.

Outline and Teacher. We use the same outlines as for the program synthesis task. However, we focus on first-order formulas, i.e., we set $a = 0$. Moreover, to find minimal formulas we iterate through k, and for each k we range l from 1 to $k + 1$. The teacher is simple: given a formula φ it checks if $A \models \varphi$ for all $A \in \mathcal{P}$, and if not, it returns (A, true) for some $A \not\models \varphi$. Then, it checks if $A \not\models \varphi$ for all $A \in \mathcal{N}$ and returns (A, false)[9] if this is not the case for some $A \models \varphi$.

Results. We substituted our SAT-based learner for the procedure for computing distinguishing formulas used in [10]. Comparing the results, the SAT-based approach appears to offer significantly better performance.

	Breakthrough	Connect4	Gomoku	Pawn-Whopping
Original system	113s	33s	11s	936s
SAT-based system	5s	16s	5s	131s

[8] E.g., isomorphism of planar graphs, bounded-degree graphs, and graphs excluding a minor; k-SAT and k-coloring are NL-complete for $k = 2$ and NP-complete for $k \geq 3$.

[9] Here, "true" and "false" are satisfied by encoded Boolean structures.

We use the same example plays for both systems – these examples were chosen by hand for the original system in [10]. But our SAT-based approach searches (faster) for formulas in more expressive logics, beyond reach for the original system. For this reason, the resulting formulas are not always correct – they are for Breakthrough and Gomoku, but not for Connect4 and Pawn-Whopping. It would be easy to overcome this by adding examples or changing the outline to match the more restrictive logics used by the original system.

5 Conclusions and Future Work

Thanks to the efficiency of modern SAT and QBF solvers, our general procedure outperforms specialized programs both in learning reductions [1] and in learning from examples [10]. We consider these early results promising and encourage further experimentation with our freely available implementation.

There are many questions we leave unanswered. For example, there is a large variance in runtime depending on the precise series of counter-examples given by the teacher. We would like to know how to choose "good" counter-examples and whether randomness [13] can help. We ask what outlines are "good", how to choose them to find the desired programs quickly and to make them readable. We are interested in re-using sub-formulas found for one problem to speed up learning in another one, and in extending our approach to quantitative settings.

References

1. Crouch, M., Immerman, N., Moss, J.E.B.: Finding reductions automatically. In: Fields of Logic and Computation. LNCS, vol. 6300, pp. 181–200 (2010)
2. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. In: Complexity of Computation, SIAM-AMS Proceedings. vol. 7, pp. 43–73 (1974)
3. Grädel, E., Kolaitis, P.G., Libkin, L., Marx, M., Spencer, J., Vardi, M.Y., Venema, Y., Weinstein, S.: Finite Model Theory and Its Applications. Springer (2007)
4. Gulwani, S.: Dimensions in program synthesis. In: Proc. PPDP 2010. pp. 13–24. New York, NY, USA (2010)
5. Immerman, N.: Relational queries computable in polynomial time. In: Proc. STOC'82. pp. 147–152 (1982)
6. Immerman, N.: Languages that capture complexity classes. SIAM J. Comput. 16(4), 760–778 (1987)
7. Immerman, N.: Descriptive Complexity. Springer-Verlag (1999)
8. Itzhaky, S., Gulwani, S., Immerman, N., Sagiv, M.: A simple inductive synthesis methodology and its applications. In: Proc. OOPSLA'10. pp. 36–46 (2010)
9. Jordan, C., Kaiser, Ł.: Experiments with reduction finding. In: Proc. SAT 2013
10. Kaiser, Ł.: Learning games from videos guided by descriptive complexity. In: Proc. AAAI 2012. pp. 963–970 (2012)
11. Kitzelmann, E.: Inductive programming: A survey of program synthesis techniques. In: AAIP 2009, Revised Papers. LNCS, vol. 5812, pp. 50–73 (2010)
12. Vardi, M.Y.: The complexity of relational query languages. In: Proc. STOC'82. pp. 137–146 (1982)
13. Zeugmann, T.: From learning in the limit to stochastic finite learning. Theoret. Comput. Sci. 364(1), 77–97 (2006)