Learning Logic Programs by Discovering Where Not to Search

Andrew Cropper and Céline Hocquette

University of Oxford

andrew.cropper@cs.ox.ac.uk, celine.hocquette@cs.ox.ac.uk

Abstract

The goal of inductive logic programming (ILP) is to search for a hypothesis that generalises training examples and background knowledge (BK). To improve performance, we introduce an approach that, before searching for a hypothesis, first discovers *where not to search*. We use given BK to discover constraints on hypotheses, such as that a number cannot be both even and odd. We use the constraints to bootstrap a constraint-driven ILP system. Our experiments on multiple domains (including program synthesis and game playing) show that our approach can (i) substantially reduce learning times by up to 97%, and (ii) scale to domains with millions of facts.

1 Introduction

The goal of inductive logic programming (ILP) (Muggleton 1991) is to search for a hypothesis (a set of rules) that generalises training examples and background knowledge (BK), where hypotheses, examples, and BK are all logic programs.

To illustrate ILP, consider learning list transformation rules with an arbitrary head literal *h*. Assume we can build rules using the unary relations *odd* and *even* and the binary relations *head* and *tail*. Then the rule space (the set of all possible rules) contains rules such as:

$$
\begin{aligned}
r_1 &= h \leftarrow \mathit{tail(A,A)} \\
r_2 &= h \leftarrow \mathit{tail(A,B),\ tail(B,A)} \\
r_3 &= h \leftarrow \mathit{tail(A,B),\ tail(B,C),\ tail(A,C)} \\
r_4 &= h \leftarrow \mathit{tail(A,A),\ head(A,B),\ odd(B)} \\
r_5 &= h \leftarrow \mathit{head(A,B),\ odd(B),\ even(B)}
\end{aligned}
$$

The hypothesis space (the set of all hypotheses) is the powerset of the rule space, so can be enormous.

To improve performance, users can impose an inductive bias (Mitchell 1997) to restrict the hypothesis space[1]. For instance, if told that *tail* is irreflexive, some systems (Law, Russo, and Broda 2014) will remove rules with the literal *tail(A,A)* from the rule space, such as r_1 and r_4. As removing a rule removes all hypotheses that contain it, a strong bias can greatly reduce the hypothesis space.

The main limitation with existing approaches is that they need a human to provide a strong bias, e.g. they need to be told that some relations are irreflexive. Furthermore, existing bias approaches, such as mode declarations (Muggleton 1995), cannot describe many useful properties, such as antitransitivity and functional dependencies (Mannila and Räihä 1994). In general, developing automatic bias discovery approaches is a grand challenge in the field (Cropper and Dumancic 2022).

To overcome this limitation, we introduce an automated bias discovery approach. The key idea is to use given BK to discover how to restrict the hypothesis space *before* searching for a solution[2]. For instance, consider the previous list transformation example. Assume we have BK with only the facts:

head(ijcai,i)	*tail(ijcai,jcai)*	*even(2)*
head(ecai,e)	*tail(ecai,cai)*	*even(4)*
head(cai,c)	*tail(jcai,cai)*	*odd(1)*
tail(ai,i)	*tail(cai,ai)*	*odd(3)*

Given this BK, if we adopt a *closed world assumption* (Reiter 1977) we can deduce that some rules will be unsatisfiable *regardless of the concept we want to learn*, i.e. regardless of specific training examples. For instance, as there is no fact of the form *tail(A,A)*, we can deduce that *tail* is irreflexive, and thus remove r_1 and r_4 from the rule space as their bodies are unsatisfiable[3]. Similarly, we can deduce that *tail* is asymmetric and antitransitive and that *odd* and *even* are mutually exclusive and thus remove rules r_2, r_3, and r_5. With this bias discovery approach, we have substantially reduced the hypothesis space before searching for a solution, i.e. we have discovered *where not to search*.

Our bias discovery approach works in two stages. First, we use the given BK to discover functional dependencies and relational properties, such as irreflexivity, asymmetry, and antitransitivity. To do so, we use a bottom-up approach (Savnik and Flach 1993) implemented in answer set programming (ASP) (Gebser et al. 2012). Second, we use the properties to build constraints to restrict the hypothesis space. For instance, if we discover that *even* and *odd* are mutually exclusive, we build constraints to prohibit rules with both the body literals

[1] All machine learning approaches need an inductive bias, i.e. bias-free learning is futile (Mitchell 1997).

[2] A *solution* is a hypothesis that generalises the examples. An *optimal* solution is the smallest solution in terms of its size in literals.

[3] These properties may only hold with the given BK. However, as the ILP problem is defined in terms of the given BK, our approach is optimally sound (Proposition 1).

odd(A) and *even(A)*. We use these constraints to bootstrap a constraint-driven ILP system (Cropper and Morel 2021). The constraints remove non-optimal hypotheses from the hypothesis space so that the system never considers them when searching for a solution.

Novelty, Impact, and Contributions. The novelty of this paper is the idea of *automatically deducing constraints about the hypothesis space before searching the hypothesis space*. As far as we are aware, this idea has not been explored before. The impact is vastly improved learning performance, demonstrated on a diverse set of tasks and domains. For instance, our approach can reduce learning times by up to 97%. Moreover, as the idea connects many AI fields, including program synthesis, constraint programming, and knowledge representation, there is much potential for broad research to build on this idea.

Overall, we make the following contributions:

- We introduce the constraint discovery problem and define *optimally sound* constraints.
- We describe a bias discovery approach that automatically discovers functional dependencies and relational properties, such as asymmetry and antitransitivity. We prove that our approach is optimally sound.
- We implement our approach in ASP and use it to bootstrap a constraint-driven ILP system.
- We experimentally show on multiple domains that our approach can (i) substantially reduce learning times by up to 97%, and (ii) scale to BK with millions of facts.

2 Related Work

Program synthesis. The goal of program synthesis is to automatically generate computer programs from examples. This topic, which Gulwani et al. (2017) consider the holy grail of AI, interests a broad community (Evans and Grefenstette 2018; Ellis et al. 2018). Although our bias discovery idea could be applied to any form of program synthesis, we focus on ILP because it induces human-readable relational programs, often from small numbers of training examples (Cropper and Dumancic 2022). Moreover, the logical representation naturally supports declarative knowledge in the form of logical constraints.

ILP. Many systems allow a human to manually specify conditions for when a rule cannot be in a hypothesis (Muggleton 1995; Srinivasan 2001; Law, Russo, and Broda 2014). Most systems only reason about the conditions *after* constructing a hypothesis, such as Aleph's rule pruning mechanism. By contrast, we automatically discover constraints and remove rules that violate them from the hypothesis space *before* searching for a hypothesis.

Constraints. Many systems use constraints to restrict the hypothesis space (Corapi, Russo, and Lupu 2011; Inoue, Doncescu, and Nabeshima 2013; Ahlgren and Yuen 2013; Kaminski, Eiter, and Inoue 2019; Cropper and Morel 2021). For instance, the Apperception (Evans et al. 2021) engine has several built-in constraints, such as a *unity condition*, which requires that objects are connected via chains of binary relations. By contrast, we automatically discover constraints before searching for a hypothesis.

Bottom clauses. Many systems use mode declarations to build bottom clauses (Muggleton 1995) to bound the hypothesis space. Bottom clauses can be seen as informing an ILP system where to search. Our approach is similar, as it restricts the hypothesis space. However, bottom clauses are example specific. To find a rule to cover an example, a learner constructs the bottom clause for that specific example, which it uses to bias the search. By contrast, our bias discovery approach is task independent and only uses the BK, not the training examples. Because of this difference, we can reuse any discovered bias across examples and tasks. For instance, if we discover that the successor relation (*succ*) is asymmetric, we can reuse this bias across multiple tasks. In addition, because of our two-stage approach, we can amortise the cost of discovering BK constraints across tasks.

Bias discovery. McCreath and Sharma (1995) automatically deduce mode declarations from the BK, such as types and whether arguments should be ground. Our approach is different because, as we use constraints, we can reason about properties that modes cannot, such as antitransitivity, functional dependencies, and mutual exclusivity. Bridewell and Todorovski (2007) learn structural constraints over the hypothesis space in a multi-task setting. By contrast, we discover biases before solving any task.

Constraint induction. Inducing constraints is popular in AI (De Raedt, Passerini, and Teso 2018). In ILP, inducing constraints has been widely studied, notably by clausal discovery approaches (De Raedt and Dehaspe 1997). These approaches induce constraints to include in a hypothesis to eliminate models. By contrast, we do not include constraints in hypotheses. Instead, we discover constraints to prune the hypothesis space.

Preprocessing. Our discovery approach is a form of *preprocessing*, which has been widely studied in AI, notably to reduce the size of a SAT instance (Eén and Biere 2005). Other preprocessing approaches in ILP focus on reducing the size of BK (Dumančić et al. 2019) or predicate invention (Hocquette and Muggleton 2020). By contrast, we discover constraints in the BK to prune the hypothesis space.

Other work. Our approach is related to automated constraint generation in constraint programming (Charnley, Colton, and Miguel 2006), finding unsatisfiable cores in SAT (Lynce and Silva 2004), and condensed representations in frequent pattern mining (De Raedt and Ramon 2004).

3 Problem Setting

We formulate our approach in the ILP learning from entailment setting (De Raedt 2008). We assume familiarity with logic programming (Lloyd 2012) and ASP (Gebser et al. 2012). The only clarification is that by *constraint* we mean a Horn clause without a positive literal.

3.1 ILP Problem

We define an ILP input. We restrict hypotheses and BK to definite programs.

Definition 1 (**ILP input**). An ILP input is a tuple $(E^+, E^-, B, \mathcal{H})$ where E^+ and E^- are sets of facts denoting positive and negative examples respectively, B is BK, and $\mathcal{H}$ is a hypothesis space, i.e a set of possible hypotheses.

We define an ILP solution:

Definition 2 (**ILP solution**). Given an ILP input $(E^+, E^-, B, \mathcal{H})$, a hypothesis $H \in \mathcal{H}$ is a *solution* when it is *complete* ($\forall e \in E^+,\ B \cup H \models e$) and *consistent* ($\forall e \in E^-,\ B \cup H \not\models e$).

Let *cost* : $\mathcal{H} \mapsto \mathbb{R}$ be an arbitrary function that measures the cost of a hypothesis. We define an *optimal* solution:

Definition 3 (**Optimal solution**). Given an ILP input $(E^+, E^-, B, \mathcal{H})$, a hypothesis $H \in \mathcal{H}$ is *optimal* when (i) H is a solution, and (ii) $\forall H' \in \mathcal{H}$, where H' is a solution, $cost(H) \leq cost(H')$.

In this paper, our cost function is the number of literals in a hypothesis. In addition, we use the notion of a *task* to refer to the problem of finding an optimal solution for an ILP input.

3.2 Constraint Discovery Problem

We denote a set of possible constraints as $\mathcal{C}$. A hypothesis $H \in \mathcal{H}$ is *consistent* with $C \subseteq \mathcal{C}$ if it does not violate any constraint in C. We denote the subset of $\mathcal{H}$ consistent with C as $\mathcal{H}_C$. We define the *constraint discovery* input:

Definition 4 (**Constraint discovery input**). A constraint discovery input is a tuple $(E^+, E^-, B, \mathcal{H}, \mathcal{C})$ where $(E^+, E^-, B, \mathcal{H})$ is an ILP input and $\mathcal{C}$ is a set of possible constraints.

We define the *constraint discovery* problem:

Definition 5 (**Constraint discovery problem**). Given a constraint discovery input $(E^+, E^-, B, \mathcal{H}, \mathcal{C})$, the *constraint discovery* problem is to find $C \subseteq \mathcal{C}$ such that $|\mathcal{H}_C| < |\mathcal{H}|$.

One might assume we want to discover *sound* constraints:

Definition 6 (**Sound constraints**). Let $I = (E^+, E^-, B, \mathcal{H}, \mathcal{C})$ be a constraint discovery input. Then $C \subseteq \mathcal{C}$ is *sound* if and only if $\forall H \in \mathcal{H}$ if H is a solution for I then $H \in \mathcal{H}_C$.

However, we often want to eliminate non-optimal solutions from the hypothesis space. For instance, consider learning to recognise lists with a single element and the hypothesis:

$$\begin{array}{l} f(A) \leftarrow length(A,B),\ one(B),\ two(B) \\ f(A) \leftarrow length(A,B),\ one(B) \end{array}$$

This hypothesis is a solution but is not optimal. We would prefer to learn an optimal solution, such as:

$$f(A) \leftarrow length(A,B),\ one(B)$$

We, therefore, define *optimally sound* constraints:

Definition 7 (**Optimally sound constraints**). Let $I = (E^+, E^-, B, \mathcal{H}, \mathcal{C})$ be a constraint discovery input. Then $C \subseteq \mathcal{C}$ is *optimally sound* if and only if $\forall H \in \mathcal{H}$ if H is an optimal solution for I then $H \in \mathcal{H}_C$.

In the next section we present an approach that discovers optimally sound constraints using the BK.

4 BK Constraint Discovery

Our approach works in two stages. First, we use BK to identify relational properties and functional dependencies. Second, we use the properties to build constraints on hypotheses to bootstrap an ILP system.

4.1 Properties

Table 1 shows the properties we consider. We generalise the properties, except antitransitive and antitriangular, to higher arities. For instance, if a ternary relation *p* is in the BK, we consider a ternary irreflexive constraint $\leftarrow$ *p(A,A,A)*. Similarly, we also identify higher-arity functional dependencies (Mannila and Räihä 1994). For instance, for the relation *append(Head,Tail,List)* we can determine that the third argument is functionally dependent on the first two. The appendix describes the properties we consider and their generalisations to arity 3.

Property Identification Rather than requiring a user to specify which properties in Table 1 hold for BK relations, we automatically discover this information. There are many efficient algorithms for discovering data dependencies (Papenbrock et al. 2015). However, as far as we are aware, no single algorithm can capture all the properties in Table 1. We, therefore, implement a bottom-up approach (Savnik and Flach 1993) in ASP. The idea is to try to find a counter-example for each property. For instance, for a binary relation *p* to be irreflexive there cannot be a counter-example *p(a,a)*. To implement this idea, we *encapsulate* all the relations in the BK, restricted to a user-specified set that may appear in a hypothesis. Specifically, for each relation *p* with arity a we add this rule to the BK:

$$holds(p, (X_1, X_2, \ldots, X_a)) \leftarrow p(X_1, X_2, \ldots, X_a)$$

We then deduce properties with ASP programs. For instance, we deduce asymmetry for binary relations by finding an answer set (a stable model) of the program:

asymmetric(P) $\leftarrow$ *holds(P,(_,_)), not non_asymmetric(P)*
non_asymmetric(P) $\leftarrow$ *holds(P,(A,B)), holds(P,(B,A))*

Likewise, we deduce that two relations P and Q are mutually exclusive with the program:

exclusive(P,Q) $\leftarrow$ *holds(P,_), holds(Q,_), not both_hold(P,Q)*
both_hold(P,Q) $\leftarrow$ *holds(P,Args), holds(Q,Args)*

We deduce that a binary relation P is functional with the program:

functional(P) $\leftarrow$ *holds(P,(_,_)), not non_functional(P)*
non_functional(P) $\leftarrow$ *holds(P,(A,B)), holds(P,(A,C)), B!=C*

The appendix includes all the ASP programs we consider.

4.2 Constraints

The output of stage one is a set of properties that hold for background relations. If a property holds for a relation, we generate the corresponding constraint to prohibit hypotheses that violate the constraint. Although these constraints can potentially be used by any ILP system, we implement our

Name	Property	Constraint	Example
Irreflexive	$\neg p(A,A)$	$\leftarrow p(A,A)$	$\leftarrow brother(A,A)$
Antitransitive	$p(A,B),\ p(B,C) \rightarrow \neg p(A,C)$	$\leftarrow p(A,B),\ p(B,C),\ p(A,C)$	$\leftarrow succ(A,B),\ succ(B,C),\ succ(A,C)$
Antitriangular	$p(A,B),\ p(B,C) \rightarrow \neg p(C,A)$	$\leftarrow p(A,B),\ p(B,C),\ p(C,A)$	$\leftarrow tail(A,B),\ tail(B,C),\ tail(C,A)$
Injective	$p(A,B),\ p(C,B) \rightarrow A{=}C$	$\leftarrow p(A,B),\ p(C,B),\ A{\neq}C$	$\leftarrow succ(A,B),\ succ(C,B),\ A{\neq}C$
Functional	$p(A,B),\ p(A,C) \rightarrow B{=}C$	$\leftarrow p(A,B),\ p(A,C),\ B{\neq}C$	$\leftarrow length(A,B),\ length(A,C),\ B{\neq}C$
Asymmetric	$p(A,B) \rightarrow \neg p(B,A)$	$\leftarrow p(A,B),\ p(B,A)$	$\leftarrow mother(A,B),\ mother(B,A)$
Exclusive	$p(A) \rightarrow \neg q(A)$	$\leftarrow p(A),\ q(A)$	$\leftarrow odd(A),\ even(A)$

Table 1: Properties and constraints. We generalise the properties, except antitransitive and antitriangular, to higher arities. The relation *succ/2* is the successor relation for natural numbers, such as *succ(1,2)*, *succ(2,3)*, *succ(3,4)*, etc.

approach to work with POPPER (Cropper and Morel 2021). POPPER is a natural choice because it frames the ILP problem as a constraint satisfaction problem. Moreover, it learns recursive programs, supports predicate invention, and is open-source[4]. We describe POPPER and our modification named DISCO.

POPPER POPPER takes as input BK, training examples, and a maximum hypothesis size and learns hypotheses as definite programs. POPPER starts with an ASP program $\mathcal{P}$ which can be viewed as a *generator* program because each model (answer set) of $\mathcal{P}$ represents a hypothesis. POPPER uses a meta-language formed of head (*h_lit/3*) and body (*b_lit/3*) literals to represent hypotheses. The first argument of each literal is the rule id, the second is the predicate symbol, and the third is the literal variables, where *0* represents *A*, *1* represents *B*, etc. For instance, POPPER represents the rule $last(A,B) \leftarrow tail(A,C),\ head(C,B)$ as the set $\{h\_lit(0,last,(0,1)),\ b\_lit(0,tail,(0,2)),\ b\_lit(0,head,(2,1))\}$. A hypothesis constraint in POPPER is a constraint written in its meta-language. For instance, the constraint $\leftarrow h\_lit(R,last,(0,1)),\ b\_lit(R,last,(1,0))$ prunes rules that contain the head literal *last(A,B)* and the body literal *last(B,A)*.

POPPER uses a generate, test, and constrain loop to search for a solution. In the generate stage, it uses an ASP solver to find a model of $\mathcal{P}$. If there is a model, POPPER converts it to a hypothesis and tests it on the examples; otherwise, it increments the hypothesis size and loops again. If a hypothesis is not a solution, POPPER builds hypothesis constraints and adds them to $\mathcal{P}$ to eliminate models and thus prunes the hypothesis space. For instance, if a hypothesis does not entail all the positive examples, POPPER builds a specialisation constraint to prune more specific hypotheses. This loop repeats until POPPER finds an optimal solution or there are no more hypotheses to test.

DISCO We augment POPPER with the ability to use the constraints from our discovery approach. The input from the user is the same as for POPPER except that we require the BK to be a Datalog program. In other words, facts and rules are allowed but not function symbols. We call this augmented version DISCO. We condition the constraints to only apply to a relation *p* if a property holds for *p*. For instance, we add an asymmetric constraint to DISCO:

$$\leftarrow asymmetric(P),\ b\_lit(R,P,(A,B)),\ b\_lit(R,P,(B,A))$$

If *asymmetric(mother)* holds, DISCO builds the constraint:

$$\leftarrow b\_lit(R,mother,(A,B)),\ b\_lit(R,mother,(B,A))$$

This constraint prunes all models that contain the literals *b_lit(R,mother,(A,B))* and *b_lit(R,mother,(B,A))*, i.e. all rules with the body literals *mother(A,B)* and *mother(B,A)*. This constraint applies to all variable substitutions for *A* and *B* and all rules *R*. For instance, the constraint prunes the rule:

$$h \leftarrow sister(A,B),\ sister(B,C),\ mother(C,D),\ mother(D,C)$$

Likewise, we add an exclusivity constraint to DISCO:

$$\leftarrow exclusive(P,Q),\ b\_lit(R,P,Vars),\ b\_lit(R,Q,Vars)$$

For instance, if *exclusive(odd,even)* holds, DISCO builds the constraint:

$$\leftarrow b\_lit(R,odd,Vars),\ b\_lit(R,even,Vars)$$

We add a functional constraint to DISCO:

$$\leftarrow functional(P),\ b\_lit(R,P,(A,B)),\ b\_lit(R,P,(A,C)),\ C!{=}B$$

For instance, if *functional(tail)* holds, DISCO builds the constraint:

$$\leftarrow b\_lit(R,tail,(A,B)),\ b\_lit(R,tail,(A,C)),\ C!{=}B$$

The ASP encodings for all the constraints are in the appendix.

To avoid complications with recursion, we do not use head predicate symbols (those in the examples) when discovering properties from the BK.

4.3 Optimal Soundness

We now prove that our approach only builds optimally sound constraints, i.e. it will not remove optimal solutions from the hypothesis space. We first show the following lemma:

Lemma 1. Each property in Table 1 has an associated constraint with an unsatisfiable body.

Proof. Follows from rewriting each property and the universal quantification. □

We show the main result:

Proposition 1 (Optimally sound constraint discovery). Given the properties in Table 1, our approach builds optimally sound constraints.

[4]ILASP (Law, Russo, and Broda 2014) is an alternative system but is closed-source and thus difficult to adapt. HEXMIL (Kaminski, Eiter, and Inoue 2019) is also an alternative system but requires metarules (program templates) as input and is thus largely restricted to dyadic logic. POPPER, by contrast, does not need metarules.

Proof. Let $H \in \mathcal{H} \setminus \mathcal{H}_C$. Assume H is an optimal solution. Since $H \in \mathcal{H}$ but $H \notin \mathcal{H}_C$ there must be a hypothesis constraint $C_1 \in C$ such that H violates C_1. C_1 is a constraint from Table 1 and prunes rules. Then there exists a rule $C_2 \in H$ and a substitution θ such that $C_1\theta \subset C_2$. C_1 has been built from our library of properties and thus has an unsatisfiable body according to Lemma 1. Since C_1 has an unsatisfiable body, then the body of C_2 is unsatisfiable. Thus C_2 does not change the coverage of H. Then $H \setminus C_2$ is a solution which contradicts our assumption. □

5 Experiments

To evaluate our claim that BK constraint discovery can reduce learning times, our experiments aim to answer the question:

Q1 Can BK constraint discovery reduce learning times?

To answer **Q1**, we compare the performance of POPPER[5] and DISCO (POPPER with BK constraint discovery).

To understand how much our approach can improve learning performance, our experiments aim to answer the question:

Q2 What effect does BK constraint discovery have on learning times given larger hypothesis spaces?

To answer **Q2**, we compare the performance of POPPER and DISCO on progressively larger hypothesis spaces.

To understand the scalability of our approach, our experiments aim to answer the question:

Q3 How long does our BK constraint discovery approach take given larger BK?

To answer **Q3**, we measure BK constraint discovery time on progressively larger BK.

As our approach is novel, there is no state-of-the-art to compare against, i.e. comparing DISCO against other systems will not allow us to evaluate the benefits of BK constraint discovery. We have, however, included a comparison of DISCO with other systems in the appendix, which shows that DISCO comprehensively outperforms state-of-the-art systems.

5.1 Experimental Domains

We use six domains. We briefly describe them. The appendix contains more details and example solutions.

Michalski trains. The goal is to find a hypothesis that distinguishes eastbound and westbound trains (Larson and Michalski 1977). We use four increasingly complex tasks.

IMDB. This real-world dataset (Mihalkova, Huynh, and Mooney 2007) contains relations between movies, actors, directors, gender and movie genre. We learn the binary relations *workedunder*, a more complex variant *workedwithsamegender*, and the disjunction of the two.

Chess. The task is to learn a rule for the king-rook-king (*krk*) endgame where the white king protects its rook (Hocquette and Muggleton 2020).

Zendo. Zendo is a multi-player game in which players try to identify a secret rule by building structures. We use four increasingly complex tasks.

[5]We use Popper 2.0.0 (Cropper 2022).

IGGP. The goal of *inductive general game playing* (Cropper, Evans, and Law 2020) (IGGP) is to induce rules to explain game traces from the general game playing competition (Genesereth and Björnsson 2013). We use six games: *minimal decay (md)*, *rock-paper-scissors (rps)*, *buttons*, *attrition*, *centipede*, and *coins*.

Program synthesis. We use a standard synthesis dataset (Cropper and Morel 2021)[6].

5.2 Experimental Setup

We enforce a timeout of 20 minutes per task. We measure the mean and standard error over 10 trials. We round times over one second to the nearest second. The appendix includes all the experimental details and example solutions.

Q1. We compare the performance of POPPER and DISCO on all tasks. We measure predictive accuracy and learning time. We separately measure BK constraint discovery time.

Q2. We compare the performance of POPPER and DISCO when varying the size of the hypothesis space. We vary the maximum size of a rule allowed in a hypothesis, i.e. the maximum number of literals allowed in a rule. We use the IGGP *md* task to answer this question.

Q3. We measure BK constraint discovery time on progressively larger BK. We generate BK for the synthesis tasks. The BK facts are relations between strings of a finite alphabet. For instance, the BK contains facts such as:

string((1,3,3,7)) *head((1,3,3,7),(1,))*
tail((1,3,3,7),(3,3,7)) *append((1,),(3,3,7),(1,3,3,7))*

We generate larger BK by increasing the size of the alphabet.

5.3 Experimental Results

Q1 Table 2 shows the learning times. It shows that on these datasets DISCO (i) never needs more time than POPPER, and (ii) can drastically reduce learning time. A paired t-test confirms the significance of the difference at the $p < 0.01$ level. For instance, for the *buttons* task (the appendix includes an example solution), the learning time is reduced from 686s to 25s, a **96%** reduction.

Table 3 shows that BK constraint discovery time is always less than a second, except for the synthesis tasks. For instance, for the real-world *imdb3* task, BK constraint discovery takes 0.02s yet reduces learning time from 366s to 287s, a 21% reduction.

To understand why our approach works, consider the *rps* task. Our approach quickly (0.02s) discovers that the relation *succ* is irreflexive, injective, functional, antitransitive, antitriangular, and asymmetric. The resulting constraints reduce the number of rules in the hypothesis space from 1,189,916 to 70,270. This reduction in the number of rules in turn considerably reduces the number of programs to consider. As

[6]Our constraint discovery implementation requires Datalog BK, a common restriction (Kaminski, Eiter, and Inoue 2019; Evans et al. 2021). However, the BK for the synthesis tasks is a definite program. Therefore, to discover BK constraints, we use a Datalog subset of the BK restricted to an alphabet with 10 symbols (0-9), where the BK constraint discovery time is 4s. We use the definite program BK for the learning task.

Task	Popper	Disco	Change
trains1	5 ± 0.1	4 ± 0.1	**-20%**
trains2	5 ± 0.2	4 ± 0.3	**-20%**
trains3	27 ± 0.8	22 ± 0.6	**-18%**
trains4	24 ± 0.8	20 ± 0.5	**-16%**
zendo1	8 ± 2	6 ± 1	**-25%**
zendo2	32 ± 2	31 ± 2	**-3%**
zendo3	33 ± 2	31 ± 1	**-6%**
zendo4	24 ± 3	24 ± 3	0%
imdb1	1 ± 0	1 ± 0	0%
imdb2	2 ± 0.1	2 ± 0	0%
imdb3	366 ± 23	287 ± 17	**-21%**
krk	48 ± 6	9 ± 0.6	**-81%**
rps	37 ± 1	6 ± 0.2	**-83%**
centipede	47 ± 2	9 ± 0.2	**-80%**
md	142 ± 7	13 ± 0.4	**-90%**
buttons	686 ± 109	25 ± 1	**-96%**
attrition	410 ± 20	57 ± 2	**-86%**
coins	496 ± 19	345 ± 18	**-30%**
buttons-goal	11 ± 0.2	5 ± 0.1	**-54%**
coins-goal	122 ± 6	76 ± 2	**-37%**
dropk	4 ± 0.3	3 ± 0.2	**-25%**
droplast	41 ± 3	23 ± 2	**-43%**
evens	33 ± 7	9 ± 1	**-72%**
finddup	51 ± 8	32 ± 4	**-37%**
last	4 ± 0.4	3 ± 0.2	**-25%**
len	31 ± 5	16 ± 2	**-48%**
sorted	74 ± 5	23 ± 1	**-68%**
sumlist	554 ± 122	320 ± 40	**-42%**

Table 2: Learning times in seconds. We round times over one second to the nearest second. Error is standard error.

shown in Table 4, the number of programs generated and tested is reduced from 6297 to 988, an **84%** reduction.

Table 5 shows the predictive accuracies. It shows that Disco (i) has equal or higher predictive accuracy than Popper on all the tasks, and (ii) can improve predictive accuracy. A McNemar's test confirms the significance of the difference at the $p < 0.01$ level.

There are two reasons for this accuracy improvement. First, Popper sometimes does not find a good solution within the time limit. By contrast, as there are fewer hypotheses for Disco to consider (Table 4), it sometimes finds a solution quicker. Second, as our approach is optimally sound (Proposition 1), it is guaranteed to lead to a hypothesis space that is a subset of the original one yet still contains all optimal solutions. According to the Blumer bound (Blumer et al. 1987), given two hypotheses spaces of different sizes, searching the smaller space will result in higher predictive accuracy compared to searching the larger one, assuming the target hypothesis is in both.

Q2 Table 6 shows that Disco can drastically reduce learning time as the hypothesis space grows (relative to Popper). For instance, for the *md* task with a maximum rule size of 6 the learning times of Popper and Disco are 113s and 10s respectively. With a maximum rule size of 8, Popper times out after 20 minutes, whereas Disco learns a solution in 47s.

Domain	Time
trains	0.22 ± 0.00
zendo	0.03 ± 0.00
imdb	0.02 ± 0.00
krk	0.10 ± 0.00
rps	0.02 ± 0.00
centipede	0.02 ± 0.00
md	0.01 ± 0.00
buttons	0.02 ± 0.00
attrition	0.01 ± 0.00
coins	0.03 ± 0.00
synthesis	4.00 ± 0.40

Table 3: BK constraint discovery times in seconds.

Q3 Figure 1 shows that our approach scales linearly in the size of the BK and can scale to millions of facts. For instance, for BK with around 8m facts, our approach takes around 47s.

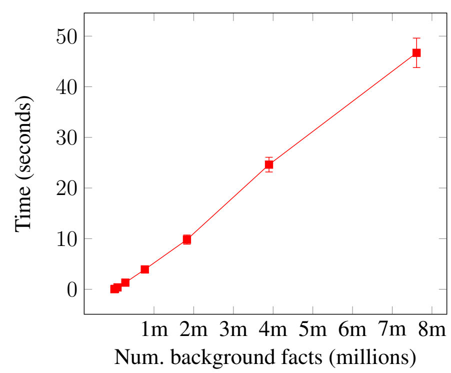

Figure 1: BK constraint discovery time when increasing the number of background facts.

6 Conclusions and Limitations

To improve learning performance, we have introduced a bias discovery approach. The three key ideas are (i) use the BK to discover a bias to restrict the hypothesis space, (ii) express the bias as constraints, and (iii) discover constraints *before* searching for a solution. Proposition 1 shows that our approach is optimally sound. Our experimental results on six domains show that our approach can (i) substantially reduce learning times, and (ii) scale to BK with millions of facts.

Limitations and Future Work

Finite BK. Our constraint discovery approach is sufficiently general to handle definite programs as BK. However, as our implementation uses ASP, we require a finite grounding of the BK. This restriction means that our implementation cannot

Task	Popper	Disco	Change
trains1	617 ± 11	575 ± 14	**-6%**
trains2	617 ± 10	556 ± 34	**-9%**
trains3	2532 ± 2	2341 ± 2	**-7%**
trains4	2712 ± 0	2519 ± 0	**-7%**
zendo1	2179 ± 783	2011 ± 698	**-7%**
zendo2	6972 ± 414	6437 ± 475	**-7%**
zendo3	7828 ± 493	7377 ± 444	**-5%**
zendo4	5512 ± 738	5303 ± 628	**-3%**
imdb1	5 ± 0	7 ± 0	+40%
imdb2	34 ± 1	39 ± 1	+14%
imdb3	330 ± 0.3	300 ± 0.7	**-9%**
krk	502 ± 59	56 ± 7	**-88%**
rps	6297 ± 7	988 ± 2	**-84%**
centipede	2312 ± 0	947 ± 0	**-59%**
md	2415 ± 51	714 ± 10	**-70%**
buttons	4610 ± 57	1248 ± 9	**-72%**
attrition	25560 ± 188	7221 ± 67	**-71%**
coins	63370 ± 1778	45037 ± 1357	**-28%**
buttons-goal	109570 ± 169	49555 ± 65	**-54%**
coins-goal	23533 ± 0	18483 ± 0	**-21%**
dropk	535 ± 34	433 ± 25	**-19%**
droplast	420 ± 17	330 ± 15	**-21%**
evens	877 ± 81	415 ± 39	**-52%**
finddup	7335 ± 919	4887 ± 641	**-33%**
last	560 ± 101	310 ± 56	**-44%**
len	1940 ± 239	1390 ± 148	**-28%**
sorted	2630 ± 159	1311 ± 105	**-50%**
sumlist	9422 ± 3121	5360 ± 2061	**-43%**

Table 4: Number of programs generated. Error is standard error.

handle BK with an infinite grounding, such as when reasoning about continuous values. Future work should address this limitation, such as by using top-down dependency discovery methods (Flach and Savnik 1999).

CWA. We adopt a closed-world assumption to discover constraints from the given BK. For instance, we assume that *odd(2)* does not hold if not given as BK. As almost all ILP systems adopt a CWA, this limitation only applies if our approach is used with a system that does not make the CWA. We also assume that the BK is noiseless, i.e. if a fact is true in the BK, then it is meant to be true. Handling noisy BK is an open challenge (Cropper and Dumancic 2022) that is beyond the scope of this paper.

Relational properties. We use a predefined set of relational properties and dependencies. The main direction for future work, therefore, is to discover more general properties and constraints. For instance, consider the two rules *h* ← *empty(A), head(A,B)* and *h* ← *empty(A), tail(A,B)*. The bodies of these rules are unsatisfiable because an empty list cannot have a head or a tail. We cannot, however, currently capture this information. Therefore, we think that this paper raises two research challenges of (i) identifying more general properties, and (ii) developing approaches to efficiently discover properties.

Task	Popper	Disco	Change
trains1	100 ± 0	100 ± 0	0%
trains2	98 ± 0	98 ± 0	0%
trains3	99 ± 0	99 ± 0	0%
trains4	100 ± 0	100 ± 0	0%
zendo1	99 ± 0	99 ± 0	0%
zendo2	96 ± 1	97 ± 1	**+1%**
zendo3	93 ± 1	93 ± 2	0%
zendo4	97 ± 0	97 ± 0	0%
imdb1	100 ± 0	100 ± 0	0%
imdb2	100 ± 0	100 ± 0	0%
imdb3	100 ± 0	100 ± 0	0%
krk	99 ± 0	99 ± 0	0%
rps	100 ± 0	100 ± 0	0%
centipede	100 ± 0	100 ± 0	0%
md	100 ± 0	100 ± 0	0%
buttons	100 ± 0	100 ± 0	0%
attrition	98 ± 0	98 ± 0	0%
coins	100 ± 0	100 ± 0	0%
buttons-goal	98 ± 1	99 ± 0	**+1%**
coins-goal	100 ± 0	100 ± 0	0%
dropk	100 ± 0	100 ± 0	0%
droplast	100 ± 0	100 ± 0	0%
evens	100 ± 0	100 ± 0	0%
finddup	98 ± 0	99 ± 0	**+1%**
last	100 ± 0	100 ± 0	0%
len	100 ± 0	100 ± 0	0%
sorted	97 ± 2	97 ± 2	0%
sumlist	90 ± 6	100 ± 0	**+11%**

Table 5: Predictive accuracies. We round times over one second to the nearest second. Error is standard error.

Task	Size	Popper	Disco	Change
md	5	12 ± 0.9	3 ± 0.3	**-75%**
md	6	113 ± 2	10 ± 0.1	**-91%**
md	7	864 ± 156	23 ± 0.9	**-97%**
md	8	*timeout*	47 ± 2	**-96%**
md	9	*timeout*	48 ± 3	**-96%**
md	10	*timeout*	52 ± 0.1	**-95%**

Table 6: Learning times when progressively increasing the maximum rule size and thus the hypothesis space. The timeout is 20 minutes (1200s). We round times over one second to the nearest second. Error is standard error.

Code, Data, and Appendices

A longer version of this paper with the appendices is available at https://arxiv.org/pdf/2202.09806.pdf. The experimental code and data are available at https://github.com/logic-and-learning-lab/aaai23-disco.

Acknowledgements

The first author is supported by the EPSRC fellowship *The Automatic Computer Scientist* (EP/V040340/1). The second author is supported by the EPSRC grant *Explainable Drug Design*. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

References

Ahlgren, J.; and Yuen, S. Y. 2013. Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Machine Learning Res.*, 14(1): 3649–3682.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's Razor. *Inf. Process. Lett.*, 24(6): 377–380.

Bridewell, W.; and Todorovski, L. 2007. Learning declarative bias. In *International Conference on Inductive Logic Programming*, 63–77. Springer.

Charnley, J. W.; Colton, S.; and Miguel, I. 2006. Automatic Generation of Implied Constraints. In *ECAI 2006*.

Corapi, D.; Russo, A.; and Lupu, E. 2011. Inductive Logic Programming in Answer Set Programming. In *ILP 2011*.

Cropper, A. 2022. Learning programs by combining programs. *CoRR*, abs/2206.01614.

Cropper, A.; and Dumancic, S. 2022. Inductive Logic Programming At 30: A New Introduction. *J. Artif. Intell. Res.*, 74: 765–850.

Cropper, A.; Evans, R.; and Law, M. 2020. Inductive general game playing. *Mach. Learn.*, 109(7): 1393–1434.

Cropper, A.; and Morel, R. 2021. Learning programs by learning from failures. *Mach. Learn.*, 110(4): 801–856.

De Raedt, L. 2008. *Logical and relational learning*. ISBN 978-3-540-20040-6.

De Raedt, L.; and Dehaspe, L. 1997. Clausal Discovery. *Mach. Learn.*, 26(2-3): 99–146.

De Raedt, L.; Passerini, A.; and Teso, S. 2018. Learning Constraints From Examples. In *AAAI-18*, 7965–7970.

De Raedt, L.; and Ramon, J. 2004. Condensed Representations for Inductive Logic Programming. In *KR2004*, 438–446.

Dumančić, S.; Guns, T.; Meert, W.; and Blockeel, H. 2019. Learning Relational Representations with Auto-encoding Logic Programs. In *IJCAI 2019*, 6081–6087.

Eén, N.; and Biere, A. 2005. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT 2005*, volume 3569, 61–75.

Ellis, K.; Morales, L.; Sablé-Meyer, M.; Solar-Lezama, A.; and Tenenbaum, J. 2018. Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Induction. In *NeurIPS 2018*, 7816–7826.

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

Evans, R.; Hernández-Orallo, J.; Welbl, J.; Kohli, P.; and Sergot, M. J. 2021. Making sense of sensory input. *Artif. Intell.*, 293: 103438.

Flach, P. A.; and Savnik, I. 1999. Database Dependency Discovery: A Machine Learning Approach. *AI Commun.*, 12(3): 139–160.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*.

Genesereth, M. R.; and Björnsson, Y. 2013. The International General Game Playing Competition. *AI Magazine*, 34(2): 107–111.

Gulwani, S.; Polozov, O.; Singh, R.; et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2): 1–119.

Hocquette, C.; and Muggleton, S. H. 2020. Complete Bottom-Up Predicate Invention in Meta-Interpretive Learning. In *IJCAI 2020*, 2312–2318.

Inoue, K.; Doncescu, A.; and Nabeshima, H. 2013. Completing causal networks by meta-level abduction. *Mach. Learn.*, 91(2): 239–277.

Kaminski, T.; Eiter, T.; and Inoue, K. 2019. Meta-Interpretive Learning Using HEX-Programs. In *IJCAI 2019*, 6186–6190.

Larson, J.; and Michalski, R. S. 1977. Inductive inference of VL decision rules. *SIGART Newsletter*, 63: 38–44.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive Learning of Answer Set Programs. In *JELIA 2014*.

Lloyd, J. W. 2012. *Foundations of logic programming*. Springer Science & Business Media.

Lynce, I.; and Silva, J. P. M. 2004. On Computing Minimum Unsatisfiable Cores. In *SAT 2004*.

Mannila, H.; and Räihä, K. 1994. Algorithms for Inferring Functional Dependencies from Relations. *Data Knowl. Eng.*, 12(1): 83–99.

McCreath, E.; and Sharma, A. 1995. Extraction of Meta-Knowledge to Restrict the Hypothesis Space for ILP Systems. In *Eighth Australian Joint Conference on Artificial Intelligence*, 75–82.

Mihalkova, L.; Huynh, T.; and Mooney, R. J. 2007. Mapping and revising markov logic networks for transfer learning. In *Aaai*, volume 7, 608–614.

Mitchell, T. M. 1997. *Machine learning*. McGraw Hill xseries in computer science. McGraw-Hill.

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

Muggleton, S. 1995. Inverse Entailment and Progol. *New Generation Comput.*, 13(3&4): 245–286.

Papenbrock, T.; Ehrlich, J.; Marten, J.; Neubert, T.; Rudolph, J.; Schönberg, M.; Zwiener, J.; and Naumann, F. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *VLDB*.

Reiter, R. 1977. On Closed World Data Bases. In *Logic and Data Bases, Symposium on Logic and Data Bases*, 55–76.

Savnik, I.; and Flach, P. A. 1993. Bottom-up induction of functional dependencies from relations. In *Proceedings of the AAAI-93 Workshop on Knowledge Discovery in Databases*, 174–185.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*.