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 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 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:

\[ r_1 = h \leftarrow \text{tail}(A,A) \]
\[ r_2 = h \leftarrow \text{tail}(A,B), \text{tail}(B,A) \]
\[ r_3 = h \leftarrow \text{tail}(A,B), \text{tail}(B,C), \text{tail}(A,C) \]
\[ r_4 = h \leftarrow \text{tail}(A,A), \text{head}(A,B), \text{odd}(B) \]
\[ r_5 = h \leftarrow \text{head}(A,B), \text{odd}(B), \text{even}(B) \]

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. 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. For instance, consider the previous list transformation example. Assume we have BK with only the facts:

\[ \text{head}(ijcai,i), \text{tail}(ijcai,ija) \text{ even}(2) \]
\[ \text{head}(ecai,e), \text{tail}(ecai,cai) \text{ even}(4) \]
\[ \text{head}(cai,c), \text{tail}(caicai) \text{ odd}(1) \]
\[ \text{tail}(ai,i), \text{tail}(caiai) \text{ odd}(3) \]

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. 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.
odd(A) and event(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 (suc) 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. McCreaeth 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 \(\text{cost} : \mathcal{H} \mapsto \mathbb{R}\) 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, \(\text{cost}(H) \leq \text{cost}(H')\).

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 \(C\). A hypothesis \(H \in \mathcal{H}\) is consistent with \(C \subseteq C\) 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 \((E^+, E^-, B, \mathcal{H}, C)\) where \((E^+, E^-, B, \mathcal{H})\) is an ILP input and \(C\) is a set of possible constraints.

We define the constraint discovery problem:

Definition 5 (Constraint discovery problem). Given a constraint discovery input \((E^+, E^-, B, \mathcal{H}, C)\), 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 \(I = (E^+, E^-, B, \mathcal{H}, C)\) be a constraint discovery input. Then \(C \subseteq \mathcal{C}\) is sound if and only if \(\forall H \in \mathcal{H} \text{ if } H\) 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:

\[
\begin{align*}
f(A) &\leftarrow \text{length}(A,B), \text{one}(B), \text{two}(B) \\
f(A) &\leftarrow \text{length}(A,B), \text{one}(B)
\end{align*}
\]

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

\[
f(A) \leftarrow \text{length}(A,B), \text{one}(B)
\]

We, therefore, define optimally sound constraints:

Definition 7 (Optimally sound constraints). Let \(I = (E^+, E^-, B, \mathcal{H}, C)\) be a constraint discovery input. Then \(C \subseteq \mathcal{C}\) is optimally sound if and only if \(\forall H \in \mathcal{H} \text{ if } H\) 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.
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. 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 \( P \) which can be viewed as a generator program because each model (answer set) of \( P \) 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 \( \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \) as the set \( \{ h_{lit}(0,\text{last},(0,1)), b_{lit}(0,\text{tail},(0,2)), b_{lit}(0,\text{head},(2,1)) \} \).

A hypothesis constraint in POPPER is a constraint written in its meta-language. For instance, the constraint \( h \leftarrow b_{lit}(R,\text{last},(1,1)), b_{lit}(R,\text{last},(1,0)) \) prunes rules that contain the head literal \( \text{last}(A,B) \) and the body literal \( \text{last}(B,A) \).

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 \( P \). 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 \( P \) 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:

\[
\text{← asymmetric}(P), b_{lit}(R,\text{mother},(A,B)), b_{lit}(R,P,(B,A))
\]

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

\[
\text{← } b_{lit}(R,\text{mother},(A,B)), b_{lit}(R,\text{mother},(B,A))
\]

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

\[
\text{← } b_{lit}(R,\text{mother},(A,B)), b_{lit}(R,\text{mother},(B,A))
\]

Likewise, we add an exclusivity constraint to DISCO:

\[
\text{← exclusive}(P,Q), b_{lit}(R,P,(B,A)), b_{lit}(R,Q,Vars)
\]

We add a functional constraint to DISCO:

\[
\text{← functional}(P), b_{lit}(R,P,(A,B)), b_{lit}(R,P,(C,A)), C!=B
\]

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

\[
\text{← } b_{lit}(R,\text{tail},(A,B)), b_{lit}(R,\text{tail},(A,C)), C!=B
\]

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.
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 $\theta$ 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. \qed

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.

**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:

\[
\text{string}((1,3,3,7)) \quad \text{head}((1,3,3,7),(1,)) \\
\text{tail}((1,3,3,7),(3,3,7)) \quad \text{append}((1,),,(3,3,7),(1,3,3,7))
\]

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 md 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, anti-triangular, 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

\(^5\)We use Popper 2.0.0 (Cropper 2022).

\(^6\)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.
A McNemar’s test confirms the significance of the difference in 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.

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.

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

<table>
<thead>
<tr>
<th>Domain</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>trains</td>
<td>0.22 ± 0.00</td>
</tr>
<tr>
<td>zendo</td>
<td>0.03 ± 0.00</td>
</tr>
<tr>
<td>imdb</td>
<td>0.02 ± 0.00</td>
</tr>
<tr>
<td>krk</td>
<td>0.10 ± 0.00</td>
</tr>
<tr>
<td>rps</td>
<td>0.02 ± 0.00</td>
</tr>
<tr>
<td>centipede</td>
<td>0.02 ± 0.00</td>
</tr>
<tr>
<td>md</td>
<td>0.01 ± 0.00</td>
</tr>
<tr>
<td>buttons</td>
<td>0.02 ± 0.00</td>
</tr>
<tr>
<td>attrition</td>
<td>0.01 ± 0.00</td>
</tr>
<tr>
<td>coins</td>
<td>0.03 ± 0.00</td>
</tr>
<tr>
<td>synthesis</td>
<td>4.00 ± 0.40</td>
</tr>
</tbody>
</table>

Table 3: BK constraint discovery times in seconds.

![Figure 1: BK constraint discovery time when increasing the number of background facts.](image)

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...
We also assume that the BK is noiseless, i.e. if a fact is true in the scope of this paper.

Future work, therefore, is to discover more general properties and dependencies. The main direction for the 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 $h ← \text{empty}(A), \text{head}(A,B)$ and $h ← \text{empty}(A), \text{tail}(A,B)$. 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.

**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


