ECAI 2024
U. Endriss et al. (Eds.)

© 2024 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA241004

Learning Logic Programs by Finding Minimal Unsatisfiable Subprograms

Andrew Cropper[*] **and Céline Hocquette**

University of Oxford

Abstract. The goal of inductive logic programming (ILP) is to search for a logic program that, with given background knowledge, generalises training examples. We introduce an ILP approach that identifies *minimal unsatisfiable subprograms* (MUSPs). We show that finding MUSPs allows us to efficiently and soundly prune the search space. Our experiments on multiple domains, including program synthesis and game playing, show that our approach can reduce learning times by 99%.

1 Introduction

The goal of inductive logic programming (ILP) is to induce a logic program (a set of logical rules) that, together with the background knowledge (BK), generalises the training examples. For instance, suppose we have BK with the relations *head/2*, *tail/2*, and *empty/1* and the following positive (E^+) and negative (E^-) examples:

$$E^+ = \{f([i,j,c,a,i],i), f([a,a,a,i],i), f([i,l,p],p)\}$$
$$E^- = \{f([i,j,c,a,i],j), f([a,a,a,i],a), f([i,l,p],i)\}$$

Given these inputs, we might want to learn a program that computes the *last* element of a list, such as:

$$\left\{ \begin{array}{l} f(A,B) \leftarrow tail(A,C), empty(C), head(A,B) \\ f(A,B) \leftarrow tail(A,C), f(C,B) \end{array} \right\}$$

To find a program, a learner (an ILP system) searches a *hypothesis space* (the set of all programs) for a *solution* (a program that correctly generalises the examples). A learner tests programs on the examples and uses the outcome to guide the search. For instance, suppose a learner tests the program:

$$\{\ f(A,B) \leftarrow head(A,B)\ \}$$

This program is too general as it entails the negative example *f([i,l,p],i)*. A learner can, therefore, prune more general programs, such as:

$$\left\{ \begin{array}{l} f(A,B) \leftarrow head(A,B) \\ f(A,B) \leftarrow tail(A,C), head(C,B) \end{array} \right\}$$

Similarly, suppose a learner tests the program:

$$h = \{\ f(A,B) \leftarrow empty(A), head(A,B), tail(A,C), head(C,B)\ \}$$

The program h is too specific because it does not entail any positive examples. A learner can, therefore, prune more specific programs. However, the simple explanation that h is too specific is crude. The program h cannot entail *anything* because an empty list cannot have a tail. In other words, the pair of literals *empty(A)* and *tail(A,C)* is unsatisfiable. Therefore, any rule that contains this pair is also unsatisfiable, so we can prune all such rules from the hypothesis space.

We argue that by finding minimal unsatisfiable programs we can prune more programs and thus improve learning performance. This idea is inspired by work in SAT on finding *minimal unsatisfiable cores* [33]. We therefore use similar terminology and say that the pair of literals *empty(A)* and *tail(A,C)* forms a *minimal unsatisfiable subprogram* (MUSP) of h as each of the literals alone is satisfiable.

Existing ILP approaches do not identify MUSPs. Some approaches reason about the failure of an entire program but cannot explain precisely *why* a program fails [27, 10, 43]. Some approaches identify erroneous rules [47, 44] or literals [38, 36] but do not identify MUSPs.

To overcome this limitation, we introduce an approach that identifies MUSPs. To explore our idea, we use the *learning from failures* (LFF) setting [10]. LFF frames the ILP problem as a constraint satisfaction problem (CSP) where each solution to the CSP represents a program. The goal of a LFF learner is to accumulate constraints to restrict the hypothesis space. We build on the LFF system POPPER [10], which can learn optimal and recursive programs from infinite domains. We enable POPPER to identify MUSPs and build constraints from them to guide subsequent search. We call our system MUSPER.

Novelty and Contributions

The main novelty of this paper is *identifying minimal unsatisfiable subprograms* and building constraints from them. The impact, which we demonstrate on multiple domains, is vastly improved learning performance. Moreover, as the idea connects many areas of AI, including machine learning, logic programming, and constraint satisfaction, we hope the idea interests a broad audience. Overall, we make the following contributions:

- We describe the *minimal unsatisfiable subprogram* problem, the problem of finding MUSPs. We show that pruning with MUSPs is optimally sound (Propositions 1 and 2) and effective (Propositions 3 and 4).
- We implement our approach in MUSPER, an ILP system which identifies the MUSPs of definite programs, including recursive programs. We prove that MUSPER always learns an optimal solution if one exists (Theorem 5).
- We experimentally show on multiple domains, including program synthesis and game playing, that our approach can substantially reduce learning times by 99%.

[*] Corresponding Author. Email: andrew.cropper@cs.ox.ac.uk

2 Related Work

Rule mining. ILP is a form of rule mining. Comparing our approach with rule mining approaches is difficult. For instance, AMIE+ [22] is a rule mining approach that uses an open-world assumption, can only use unary and binary relations, and requires facts as input. By contrast, MUSPER uses the closed-world assumption, can use any arity relations, and can learn from definite programs as BK.

ILP. Many ILP systems [37, 5, 50, 53, 16] struggle to learn recursive programs and perform predicate invention [12]. By contrast, MUSPER can learn recursive programs and perform predicate invention. In addition, unlike many ILP systems [19, 27, 48, 14, 24, 40], we do not require metarules to restrict the hypothesis space.

Rule selection. Many systems formulate the ILP problem as a rule selection problem [8, 30, 27, 44, 3]. These approaches precompute every possible rule in the hypothesis space and then search for a subset that generalises the examples. Because they precompute all possible rules, they cannot learn rules with many literals. For instance, if allowed to use the relations *odd/1* and *even/1*, ILASP4 [30] will precompute all possible rules with *odd(A)* and *even(A)* in the body. By contrast, MUSPER does not precompute all rules and instead builds constraints from MUSPs to restrict the generation of rules. For instance, if MUSPER sees a rule with *odd(A) even(A)*, it will identify that this pair of literals forms a MUSP and will henceforth never build a rule with both of these literals.

Bottom clauses. Many ILP systems, such as ALEPH, use bottom clauses [37] (or variants, such as Kernel sets [45]) to restrict the hypothesis space [37, 31]. The bottom clause of an example e is the most specific clause that entails e. Bottom clauses have limitations. Building a bottom clause can be expensive [37]. In addition, in the worst-case, a learner needs to build a bottom clause for every positive training example. Indeed, the Kernel set, as used by XHAIL [45], is built from all positive examples at once. Moreover, approaches that use bottom clauses struggle to learn programs with recursion or predicate invention. By contrast, our approach does not use bottom clauses and does not have these limitations.

Theory repair and revision. Theory repair approaches [7] revise a hypothesis by applying generalisation and specialisation operators to revision points. Theory revision approaches [52, 1, 39] can identify literals as revision points, but often with limitations, such as requiring user-interaction [47] or restrictions on the training examples [46]. By contrast, MUSPER is guaranteed to find an optimal solution if one exists without such restrictions.

Explanations. POPPER [10] builds constraints from whole programs to prune the hypothesis space. By contrast, we also build constraints from MUSPs, which leads to better pruning (Section 3.4). ILASP4 [30] and PROSYNTH [44] generate constraints from rules that do not entail an example. These two approaches precompute all possible rules and only identify erroneous rules. By contrast, we do not precompute all possible rules and identify minimal subsets of unsatisfiable literals. METAGOL [38] builds a program literal-by-literal and tests it on the examples. If a program does not entail any positive examples, METAGOL backtracks to consider alternative literals, i.e. it identifies unsatisfiable literals. METAGOL repeatedly reconsiders unsatisfiable programs, whereas MUSPER learns constraints so it never does. HEMPEL [36] analyses the execution paths of definite programs to identify unsatisfiable literals. HEMPEL analyses rules using SLD-resolution from left to right and, therefore, can only identify definite subprograms and is not guaranteed to find MUSPs. For instance, given the unsatisfiable rule *f(A,B)* ← *element(A,B),odd(B),even(B)*, HEMPEL cannot find a smaller unsatisfiable subprogram. By contrast, MUSPER identifies that the literals *odd(B)* and *even(B)* form a MUSP.

User constraints. Many ILP systems use constraints to restrict the hypothesis space [50, 26, 20]. For instance, if a user knows that two literals are unsatisfiable, they could provide this information to ILASP4 in the form of *meta-constraints* or ALEPH via user-defined pruning conditions. However, in these cases, the constraints are given as input to the system by a user. By contrast, our approach discovers constraints during the learning process.

Constraint discovery. Clausal discovery approaches [15] discover constraints to include in a hypothesis to eliminate models. By contrast, we discover constraints to prune the hypothesis space. DISCO [9] discovers constraints from the BK as a preprocessing step, which an ILP system can use to restrict the hypothesis space. DISCO uses a predefined set of constraints, such as *asymmetry* and *antitransitivity*. We differ because we (i) find MUSPs, (ii) discover constraints during the learning, not as a preprocessing step, (iii) do not use a predefined set of constraints, and (iv) jointly consider the BK and the examples. As our experiments show, these differences greatly impact learning performance.

3 Problem Setting

We describe our problem setting. We assume familiarity with logic programming [32] but have included a summary in the appendix.

3.1 Learning From Failures

We use the LFF setting. We learn definite programs with the least Herbrand model semantics. A *hypothesis* is a set of definite clauses. We use the term *program* synonymously with *hypothesis*. A *hypothesis space* $\mathcal{H}$ is a set of hypotheses. LFF uses *hypothesis constraints* to restrict the hypothesis space. Let $\mathcal{L}$ be a language that defines hypotheses. A *hypothesis constraint* is a constraint (a headless rule) expressed in $\mathcal{L}$. Let C be a set of hypothesis constraints written in a language $\mathcal{L}$. A hypothesis is *consistent* with C if when written in $\mathcal{L}$ it does not violate any constraint in C. We denote as $\mathcal{H}_C$ the set of hypotheses from $\mathcal{H}$ which are consistent with C.

We define the LFF input:

Definition 1 (LFF input). *A* LFF *input is a tuple* $(E^+, E^-, B, \mathcal{H}, C)$ *where* E^+ *and* E^- *are sets of ground atoms denoting positive and negative examples respectively;* B *is a definite program denoting background knowledge;* $\mathcal{H}$ *is a hypothesis space, and* C *is a set of hypothesis constraints.*

We define a LFF solution:

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

We define an *optimal* solution, where *size(h)* denote the number of literals in the hypothesis h:

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

If a hypothesis is not a solution then it is a *failure*. A hypothesis is *incomplete* when $\exists e \in E^+,\ h \cup B \not\models e$; *inconsistent* when $\exists e \in E^-,\ h \cup B \models e$; *partially complete* when $\exists e \in E^+,\ h \cup B \models e$; and *totally incomplete* when $\forall e \in E^+,\ h \cup B \not\models e$.

Constraints. A LFF learner builds constraints from failures to prune the hypothesis space. Cropper and Morel [10] introduce constraints based on subsumption [42]. A clause c_1 *subsumes* a clause c_2 ($c_1 \preceq c_2$) if and only if there exists a substitution θ such that $c_1\theta \subseteq c_2$. A clausal theory t_1 subsumes a clausal theory t_2 ($t_1 \preceq t_2$) if and only if $\forall c_2 \in t_2, \exists c_1 \in t_1$ such that c_1 subsumes c_2. A clausal theory t_1 is a *specialisation* of a clausal theory t_2 if and only if $t_2 \preceq t_1$. A clausal theory t_1 is a *generalisation* of a clausal theory t_2 if and only if $t_1 \preceq t_2$. There are three standard constraints for a failure hypothesis h. A *specialisation* constraint prunes specialisations of h. A *generalisation* constraint prunes generalisations of h. Cropper and Morel [11] introduce *redundancy* constraints, which are difficult to tersely describe. Although our algorithm supports predicate invention, for simplicity, in the following high-level explanation, we ignore predicate invention and focus on redundancy constraints for recursive programs. If a hypothesis h is totally incomplete then h (or a specialisation of it) cannot appear in an optimal solution h_o unless h_o has a recursive rule that does not specialise a rule in h. For instance, if a totally incomplete hypothesis has a single rule r then r can only be in an optimal solution h_o if h_o has a recursive rule that does not specialise r. Otherwise, r is redundant in h_o. If every recursive rule in a hypothesis p specialises a rule in a hypothesis q then we say that p is *redundant* with respect to q. The appendix includes examples of redundant programs.

3.2 Minimal Unsatisfiable Subprograms

We define the MUSP problem. We define a *subprogram*:

Definition 4 (Subprogram). *Let h be a hypothesis and g be a set of Horn clauses. Then g is a* subprogram *of h if and only if the following conditions all hold: (i) $size(g) < size(h)$, (ii) $g \preceq h$, and (iii) $\forall c_1 \in g, \exists c_2 \in h$ such that $c_1 \preceq c_2$.*

Example 1. Consider the program:

$$h = \left\{ \begin{array}{l} f(A,B) \leftarrow tail(A,C), empty(C), head(A,B) \\ f(A,B) \leftarrow tail(A,C), f(C,B) \end{array} \right\}$$

Then h has the subprogram:

$$\left\{ \begin{array}{l} f(A,B) \leftarrow tail(A,C), head(A,B) \\ f(A,B) \leftarrow tail(A,C), f(C,B) \end{array} \right\}$$

The following program is not a subprogram of h as it has more literals than h and thus violates condition (i):

$$\left\{ \begin{array}{l} f(A,B) \leftarrow tail(A,C), tail(C,D), head(A,B) \\ f(A,B) \leftarrow tail(A,C), empty(C), head(A,B) \\ f(A,B) \leftarrow tail(A,C), f(C,B) \end{array} \right\}$$

The following program is not a subprogram of h as it does not subsume h and thus violates condition (ii):

$$\left\{ f(A,B) \leftarrow tail(A,C), empty(C), head(A,B) \right\}$$

The following program is not a subprogram of h as its second rule does not subsume a rule in h and thus it violates condition (iii):

$$\left\{ \begin{array}{l} f(A,B) \leftarrow tail(A,C) \\ f(A,B) \leftarrow empty(A), tail(A,C) \end{array} \right\}$$

We define an *unsatisfiable subprogram*:

Definition 5 (Unsatisfiable subprogram). *Let $(E^+, E^-, B, \mathcal{H}, C)$ be a LFF input, $h \in \mathcal{H}_C$ be a hypothesis, g be a subprogram of h, and g be totally incomplete with respect to E^+ and B. Then g is an* unsatisfiable subprogram *of h.*

Example 2. Consider the positive examples $E^+ = \{f([],0), f([i,j,c,a,i],5)\}$, an appropriate BK containing the relations *head/2*, *tail/2*, *increment/2*, *zero/1*, and *empty/1* and the hypothesis:

$$h = \left\{ f(A,B) \leftarrow empty(A), head(A,B), tail(A,C), head(C,B) \right\}$$

Assuming standard definitions for the relations, two unsatisfiable subprograms of h are:

$$\left\{ f(A,B) \leftarrow empty(A), head(A,B) \right\}$$
$$\left\{ \leftarrow empty(A), head(A,B), tail(A,C) \right\}$$

Note that a subprogram can contain headless rules (goal clauses) as in the second subprogram above.

We define a *minimal unsatisfiable subprogram*:

Definition 6 (Minimal unsatisfiable subprogram). *Let S be the set of all unsatisfiable subprograms of the hypothesis h. Then $g \in S$ is a* minimal unsatisfiable subprogram *(MUSP) of h if and only if $\forall g' \in S, size(g) \leq size(g')$.*

Example 3. Reusing h, the BK, and E^+ from Example 2, h has the MUSPs:

$$\left\{ \leftarrow empty(A), head(A,B) \right\}$$
$$\left\{ \leftarrow empty(A), tail(A,C) \right\}$$
$$\left\{ f(A,B) \leftarrow head(A,B) \right\}$$

A MUSP differs to a minimal unsatisfiable core [33], a minimal unsatisfiable subset [2], and an *inconsistent core* [21], which are all usually defined in terms of subset minimality. By contrast, a MUSP is defined in terms of cardinality minimality, which refers to the number of literals in a hypothesis.

We define the *MUSP problem*:

Definition 7 (MUSP problem). *The* MUSP problem *is to find all MUSPs of a hypothesis.*

Section 4 describes MUSPER, which solves the MUSP problem.

3.3 MUSP Constraints

We want to identify MUSPs to build constraints from them to prune the hypothesis space of an ILP learner. We describe the constraints that we can build from a MUSP.

Since a MUSP is totally incomplete then its specialisations cannot be solutions:

Proposition 1 (MUSP specialisations). *Let h be a program, m be a MUSP of h, and h' be a specialisation of m. Then h' is not a solution.*

Proof. Since m is totally incomplete and h' is a specialisation of m then h' is totally incomplete. Therefore h' is not a solution. □

We show an example of this constraint:

Example 4. Consider the positive example E^+ = *{f([x],y)}* and the hypothesis:

$$h = \left\{ f(A,B) \leftarrow tail(A,C), tail(C,A), head(C,B) \right\}$$

A MUSP of h is:

$$m = \quad \{ \ \leftarrow tail(A,C),\ tail(C,A) \ \}$$

We can prune the specialisations of m, such as:

$$\{ \ f(A,B) \leftarrow tail(A,C),\ tail(C,A) \ \}$$
$$\{ \ \leftarrow tail(A,C),\ tail(C,A),\ head(A,D) \ \}$$

Since a MUSP m is totally incomplete then its specialisations cannot be in an optimal solution without a recursive rule that does not specialise m:

Proposition 2 (**MUSP redundancy**). *Let h and h' be hypotheses, m be a MUSP of h, and h' be redundant with respect to m. Then h' is not an optimal solution.*

Proof. Since m is a MUSP then it is totally incomplete. Cropper and Morel [11] show that hypotheses redundant with respect to a totally incomplete hypothesis are not optimal solutions (Theorem 1). Therefore h' is not an optimal solution. □

We show an example of this constraint:

Example 5. Consider the positive example E^+ = *{f([x],y)}* and the hypothesis:

$$h = \quad \{ \ f(A,B) \leftarrow tail(A,C),\ tail(C,D),\ head(D,B) \ \}$$

A MUSP of h is:

$$m = \quad \{ \ f(A,B) \leftarrow tail(A,C),\ tail(C,D) \ \}$$

We can prune the following hypothesis because it is redundant with respect to m:

$$\{ \ f(A,B) \leftarrow head(A,B) \ \}$$
$$\{ \ f(A,B) \leftarrow tail(A,C),\ tail(C,D),\ f(D,B) \ \}$$

In Section 4, we introduce MUSPER which uses these optimally sound constraints to prune the hypothesis space.

3.4 MUSP Benefits

We show that using MUSPs to build constraints that prune specialisations and redundant hypotheses leads to more pruning.

Proposition 3 (**Specialisation pruning**). *Let h be a hypothesis, m be a MUSP of h, S_h be the specialisations of h, and S_m be the specialisations of m. Then $S_h \subset S_m$.*

Proof. Since h specialises m and $size(h) > size(m)$ then (i) S_m includes h and S_h, and (ii) S_h does not include m. □

Proposition 4 (**Redundancy pruning**). *Let h be a hypothesis, m be a MUSP of h, R_h be the hypotheses redundant with respect to h, and R_m be the hypotheses redundant with respect to m. Then $R_h \subset R_m$.*

Proof. We show that $h' \in R_h$ implies $h' \in R_m$. Since h' is redundant with respect to h then there is no recursive rule r' in h' that does not specialise a rule in h. We show that there is no recursive rule r' in h' that does not specialise a rule in m. For contradiction, suppose there is such a rule r', which implies that no rule in m generalises r'. By condition (ii) in Definition 4, m subsumes h. Since m subsumes h then every rule in h is subsumed by a rule in m, including r', which leads to a contradiction. Therefore $h' \in R_m$. Moreover, $m \in R_m$ but $m \notin R_h$ since h specialises m and $size(m) < size(h)$. Therefore $R_h \subset R_m$, which completes the proof. □

4 Algorithm

We now describe our MUSPER approach. To delineate the novelty, we first describe POPPER.

POPPER. POPPER takes as input background knowledge (*bk*), positive (*pos*) and negative (*neg*) training examples, and a maximum hypothesis size (*max_size*). POPPER uses a generate, test, and constrain loop to find an optimal solution. POPPER starts with a CSP program $\mathcal{P}$ (hidden in the generate function). The models of $\mathcal{P}$ correspond to hypotheses (definite programs). In the generate stage, POPPER searches for a model of $\mathcal{P}$. If there is no model, POPPER increments the hypothesis size and loops again. If there is a model, POPPER converts it to a hypothesis h. In the test stage, POPPER uses Prolog to test h on the examples. If h is a solution, POPPER returns it. If h is a failure then, in the constrain stage, POPPER builds hypothesis constraints (represented as CSP constraints) from h. POPPER adds these constraints to $\mathcal{P}$ to prune models and thus prune the hypothesis space. For instance, if h is incomplete, POPPER builds a specialisation constraint to prune its specialisations. If h is inconsistent, POPPER builds a generalisation constraint to prune its generalisations. To reiterate, POPPER builds constraints from whole programs. POPPER repeats this loop until it finds an optimal solution or exhausts the models of $\mathcal{P}$.

MUSPER

MUSPER (Algorithm 1) builds on POPPER with one major difference. If a program h is totally incomplete then MUSPER calls Algorithm 2 (line 13) to identify the MUSPs of h and build constraints from them. To reiterate, the ability to identify MUSPs and build constraints from them is the main novelty and contribution of this paper.

Algorithm 1 MUSPER

```
1 def MUSPer(bk, pos, neg, max_size):
2   cons = {}
3   size = 1
4   while size ≤ max_size:
5     h = generate(cons, size)
6     if h == UNSAT:
7       size += 1
8       continue
9     outcome = test(pos, neg, bk, h)
10    if outcome == (COMPLETE, CONSISTENT)
11      return h
12    if outcome == (TOTALLY_INCOMPLETE, _)
13      cons += unsat_constraints(h, pos, bk)
14    cons += constrain(h, outcome)
15  return {}
```

4.0.1 MUSPs

Algorithm 2 builds constraints from MUSPs. To find MUSPs (line 2), we use what Dershowitz et al. [17] call a *folk algorithm*. We search for subprograms of the program h by removing a literal from it and checking that the subprogram obeys the conditions in Definition 4 (line 12). We also enforce other syntactic restrictions, such as that the literals in a rule are connected[1]. We check each subprogram to

[1] The literals in a rule are connected if they cannot be partitioned into two sets such that the variables in the literals of one set are disjoint from the variables in the literals of the other set.

Algorithm 2 Find MUSPs

```
1 def unsat_constraints(h, pos, bk):
2   MUSPs = find_MUSPs(h, pos, bk)
3   cons = {}
4   for MUSP in MUSPs:
5     cons += build_spec_con(MUSP)
6     cons += build_redund_con(MUSP)
7   return cons
8
9 def find_MUSPs(h, pos, bk):
10  MUSPs = {}
11  has_unsat_subprog = False
12  for subprog in subprogs(h):
13    if unsat(subprog, pos, bk):
14       has_unsat_subprog = True
15       MUSPs += find_MUSPs(subprog, pos, bk):
16  if has_unsat_subprog == False:
17    MUSPs += {h}
18  return MUSPs
```

see if it is unsatisfiable (line 13). If a subprogram is unsatisfiable, we recursively find the MUSPs of the subprogram itself and add the result to a set of minimal unsatisfiable subprograms (line 15). Otherwise, we add h to the set of minimal unsatisfiable subprograms (line 17).

4.0.2 Satisfiability

In our implementation of Algorithm 2 we use Prolog to check the satisfiability of a subprogram (line 13). There are two cases. If all the rules in a subprogram s have a head literal then we say that s is satisfiable if it entails at least one positive example. If a subprogram s contains a rule r that does not have a head literal then we add a *is_sat* literal to the head of r. We say that s is satisfiable if *is_sat* is true. For instance, given the subprogram:

$$\{ \leftarrow \textit{tail(A,C), tail(C,A)} \}$$

We test the program:

$$\{ \textit{is_sat} \leftarrow \textit{tail(A,C), tail(C,A)} \}$$

4.0.3 Constraints

Since a MUSP is totally incomplete then its specialisations cannot be a solution (Proposition 1). Likewise, since a MUSP is totally incomplete then we can prune programs redundant with respect to it (Proposition 2). Therefore, Algorithm 2 builds a specialisation and a redundancy constraint for each MUSP (lines 5 and 6).

4.0.4 Correctness

We show that MUSPER is correct:

Theorem 5 (MUSPER correctness). MUSPER *returns an optimal solution if one exists.*

Proof. Cropper and Morel [10] show that given optimally sound constraints POPPER returns an optimal solution if one exists (Theorem 1). MUSPER builds on POPPER by additionally pruning specialisations and redundancies of MUSPs, which are optimally sound by Propositions 1 and 2 respectively. Therefore, MUSPER never prunes optimal solutions so returns an optimal solution if one exists. □

5 Experiments

We claim that identifying MUSPs can improve the learning performance of an ILP system. Propositions 3 and 4 support this claim and show that identifying MUSPs improves pruning. However, it is unclear whether in practice the overhead of finding MUSPs and building the constraints outweigh the pruning benefits. Therefore, our experiments aim to answer the question:

Q1 Can identifying MUSPs reduce learning times?

To answer **Q1**, we compare the performance of MUSPER against POPPER. As MUSPER builds on POPPER, the only difference between the systems is the ability to identify MUSPs. Therefore, this comparison directly measures the impact of our idea.

Comparing MUSPER against other systems besides POPPER will not allow us to evaluate the idea of identifying MUSPs because it will not allow us to identify the source of empirical gains. However, to understand how MUSPER compares to other approaches, our experiments try to answer the question:

Q2 How does MUSPER compare to other approaches?

To answer **Q2**, we compare MUSPER against ALEPH [50], METAGOL [38], POPPER, and DISCO [9][2]. We use these systems because they can learn definite programs with function symbols and recursive programs.

5.1 Domains

We briefly describe our domains. We use diverse domains used by a wide variety of authors. The appendix contains more details, such as example solutions for each task and statistics about the problem sizes.

Trains. The goal is to find a hypothesis that distinguishes east and west trains [28].

Zendo. Zendo is a multiplayer game where players must discover a secret rule by building structures. Zendo has attracted interest in cognitive science [6, 41].

IMDB. We use a real-world dataset which contains relations between movies, actors, and directors [35]. This dataset is frequently used to evaluate rule learning systems [34, 18, 29].

IGGP. The goal of *inductive general game playing* [13] (IGGP) is to induce rules to explain game traces from the general game playing competition [23].

Chess. The task is to learn chess patterns in the king-rook-king (*krk*) endgame [25]. This domain uses definite clauses with function symbols in the BK.

Graph problems. We use frequently used graph problems [19, 24] which require learning recursive programs.

Program synthesis. We use a program synthesis dataset [10] augmented with 5 new tasks. These tasks all require the ability to learn recursive programs from non-factual (Prolog) data.

SQL. This dataset [48] contains 15 real-world tasks that involve synthesising SQL queries. This dataset is frequently used to evaluate synthesis systems [49, 44, 3]. As there is no testing data, we report training accuracy.

[2] We tried other systems. We considered ILASP4 [30] and ASPSynth [3]. However, both systems precompute every possible rule in a hypothesis space, which is infeasible for our datasets, and would, for instance, require precomputing 10^{15} rules for the *coins* task. In addition, both systems cannot learn definite programs with function symbols so are unusable on the *synthesis* and *krk* domains. We also tried HEXMIL [27] and LOUISE [40], both metarule-based approaches similar to METAGOL. However, their performance was considerably worse than METAGOL so we excluded them.

5.2 Experimental Setup

We enforce a timeout of 30 minutes per task. The appendix includes all the experimental details and example solutions. We measure predictive accuracy, learning time (including the time to discover MUSPs), and the number of programs considered. We also report the time taken to discover MUSPs, i.e. the overhead of our approach. We measure the mean over 10 trials. The error bars in all tables denote standard error.

5.3 Experimental Results

5.3.1 *Q1. Can Identifying MUSPs Reduce Learning Times?*

Table 1 shows the learning times aggregated per domain. Table 3 shows the learning times for tasks where the times of MUSPER and POPPER differ. These results show that MUSPER never needs more time than POPPER. A paired t-test confirms the significance of the difference ($p < 0.01$). MUSPER can drastically reduce learning time. For instance, on the *sql-06* task, POPPER times out after 1800s, whilst MUSPER takes only 14s, a 99% reduction. Likewise, on the *coins-g* task, POPPER times out after 1800s whilst MUSPER only takes 6s, a 99% reduction.

Table 2 shows the predictive accuracies aggregated per domain. MUSPER has equal or higher predictive accuracy than POPPER on all but two domains, where the difference is negligible. These results show that MUSPER can drastically reduce learning times whilst maintaining high levels of predictive accuracy.

Domain	POPPER	MUSPER	Change
trains	13 ± 1	13 ± 1	0%
zendo	36 ± 4	36 ± 4	0%
imdb	193 ± 51	148 ± 39	**–23%**
iggp	617 ± 66	207 ± 34	**–66%**
krk	41 ± 6	38 ± 5	**–7%**
graph	12 ± 5	8 ± 2	**–33%**
synthesis	343 ± 39	199 ± 33	**–41%**
sql	594 ± 63	13 ± 1	**–97%**

Table 1: Learning times (seconds).

Domain	ALEPH	METAGOL	DISCO	POPPER	MUSPER
trains	100 ± 0	45 ± 4	99 ± 0	99 ± 0	99 ± 0
zendo	93 ± 1	55 ± 2	97 ± 1	96 ± 1	97 ± 1
imdb	67 ± 4	39 ± 3	100 ± 0	100 ± 0	100 ± 0
iggp	88 ± 3	34 ± 3	99 ± 0	100 ± 0	99 ± 0
krk	95 ± 1	50 ± 0	70 ± 4	70 ± 4	73 ± 4
graph	50 ± 0	77 ± 3	98 ± 1	98 ± 1	97 ± 1
synthesis	54 ± 1	57 ± 1	97 ± 1	97 ± 1	98 ± 1
sql	100 ± 0	13 ± 3	95 ± 2	88 ± 3	100 ± 0

Table 2: Predictive accuracies (%).

Table 4 shows the aggregated number of programs considered by each system. The full table for each task is in the appendix. MUSPER can consider considerably fewer programs than POPPER. For instance, on the *sql-02* task, POPPER considers 25,533 programs whereas MUSPER only considers 52, a 99% reduction. This result shows that constraints built from MUSPs can effectively prune the hypothesis space.

Table 5 shows the proportion of the total learning time spent identifying MUSPs. The overhead is usually less than a few seconds and represents at most 23% of the learning time.

To illustrate why our approach works, consider the *last* task, where MUSPER can generate the hypothesis:

$$\{\ f(A,B) \leftarrow head(A,B),\ zero(C),\ decrement(C,B)\ \}$$

Task	POPPER	MUSPER	Change
zendo3	48 ± 6	$\mathbf{46 \pm 4}$	**-4%**
imdb3	571 ± 37	$\mathbf{435 \pm 34}$	**-23%**
md	270 ± 25	$\mathbf{53 \pm 2}$	**-80%**
buttons	743 ± 144	$\mathbf{50 \pm 4}$	**-93%**
rps	185 ± 2	$\mathbf{162 \pm 2}$	**-12%**
coins	1028 ± 71	$\mathbf{952 \pm 63}$	**-7%**
buttons-g	21 ± 0.5	$\mathbf{12 \pm 0}$	**-42%**
coins-g	*timeout*	$\mathbf{6 \pm 0}$	**-99%**
attrition	740 ± 11	$\mathbf{290 \pm 24}$	**-60%**
centipede	147 ± 2	$\mathbf{129 \pm 11}$	**-12%**
krk	41 ± 6	$\mathbf{38 \pm 5}$	**-7%**
cyclic	66 ± 27	$\mathbf{36 \pm 15}$	**-45%**
dropk	10 ± 1	$\mathbf{7 \pm 1}$	**-30%**
droplast	27 ± 2	$\mathbf{19 \pm 1}$	**-29%**
evens	58 ± 7	$\mathbf{38 \pm 3}$	**-34%**
finddup	133 ± 12	$\mathbf{74 \pm 11}$	**-44%**
last	22 ± 2	$\mathbf{11 \pm 1}$	**-50%**
contains2	306 ± 11	$\mathbf{268 \pm 12}$	**-12%**
length	57 ± 6	$\mathbf{24 \pm 5}$	**-57%**
reverse	1053 ± 149	$\mathbf{203 \pm 19}$	**-80%**
sorted	253 ± 25	$\mathbf{216 \pm 26}$	**-14%**
sumlist	276 ± 49	$\mathbf{25 \pm 1}$	**-90%**
next	59 ± 10	$\mathbf{44 \pm 7}$	**-25%**
rotateN	437 ± 43	$\mathbf{292 \pm 49}$	**-33%**
inttobin	83 ± 12	$\mathbf{65 \pm 8}$	**-21%**
chartointallodd	638 ± 149	$\mathbf{104 \pm 5}$	**-83%**
twosucc	*timeout*	$\mathbf{1764 \pm 23}$	**-2%**
sql-01	14 ± 6	$\mathbf{6 \pm 0}$	**-57%**
sql-02	936 ± 136	$\mathbf{29 \pm 1}$	**-96%**
sql-03	1126 ± 138	$\mathbf{24 \pm 2}$	**-97%**
sql-04	*timeout*	$\mathbf{71 \pm 4}$	**-96%**
sql-05	8 ± 0	$\mathbf{5 \pm 0}$	**-37%**
sql-06	*timeout*	$\mathbf{14 \pm 1}$	**-99%**
sql-07	9 ± 0	$\mathbf{6 \pm 0}$	**-33%**
sql-08	1527 ± 142	$\mathbf{8 \pm 0}$	**-99%**
sql-09	5 ± 0	$\mathbf{4 \pm 0}$	**-20%**
sql-10	11 ± 1	$\mathbf{6 \pm 0}$	**-45%**
sql-11	1615 ± 125	$\mathbf{8 \pm 0}$	**-99%**
sql-12	41 ± 11	$\mathbf{5 \pm 0}$	**-87%**

Table 3: Learning times (seconds). We only show tasks where the two approaches differ. The full table is in the appendix.

From this hypothesis, MUSPER identifies the MUSP:

$$\{\ \leftarrow zero(C),\ decrement(C,B)\ \}$$

This program is unsatisfiable because the BK for this task is defined for natural numbers, so zero has no predecessor. MUSPER prunes the specialisations and redundancies of this MUSP, thus vastly reducing the hypothesis space.

Likewise, on the *buttons* task, MUSPER sometimes generates the hypothesis:

$$\{\ next(A,B) \leftarrow my_true(A,B),\ succ(B,C),\ succ(C,B)\ \}$$

From this hypothesis, MUSPER identifies the MUSP:

$$\{\ \leftarrow succ(B,C),\ succ(C,B)\ \}$$

This program is unsatisfiable because the successor relation (*succ*) is asymmetric. MUSPER prunes any hypothesis that contains this pair of literals.

Likewise, on the *rps* (rock, paper, scissors) task, MUSPER identifies many MUSPs, including:

$$\{\ \leftarrow does(E,Player,A1),\ does(E,Player,A2),\ beats(A1,A2)\ \}$$

This program is unsatisfiable because a player (*Player*) cannot make two actions (*A1* and *A1*) where one beats another. In other words, MUSPER identifies that a player can only make one action.

Similarly, on the *length* task, MUSPER identifies the MUSP:

$$\{\ \textit{f(A,B)}\leftarrow \textit{one(B), tail(A,C), tail(C,D)}\ \}$$

This program is unsatisfiable because if a list A has length $B = 1$ then its tail C does not have a tail D.

MUSPER reduces learning times when it discovers small MUSPs early in the search but is otherwise less helpful. For instance, MUSPER does not help much on the *zendo* tasks because these tasks have many background relations that almost always hold, such as *has_piece(State, Piece)*, because every Zendo game state has a piece, and *size(Piece, Size)*, because a piece always has a size.

Overall, the results in this section show that the answer to **Q1** is yes, identifying MUSPs can drastically improve learning times whilst maintaining high predictive accuracies.

Domain	POPPER	MUSPER	Change
trains	1630 ± 160	1632 ± 159	+0%
zendo	2303 ± 264	2304 ± 261	+0%
imdb	177 ± 40	162 ± 36	**–8%**
iggp	43122 ± 6189	16623 ± 2795	**–61%**
krk	440 ± 71	409 ± 60	**–7%**
graph	103 ± 34	81 ± 26	**–21%**
synthesis	4395 ± 387	1595 ± 162	**–63%**
sql	14328 ± 1932	44 ± 4	**–99%**

Table 4: Number of programs considered.

Domain	Total	MUSPs	Ratio
trains	13 ± 1	0 ± 0	0%
zendo	36 ± 4	0.1 ± 0	0%
imdb	148 ± 39	3 ± 1	2%
iggp	207 ± 34	4 ± 0	1%
krk	38 ± 5	0.7 ± 0.1	1%
graph	8 ± 2	1 ± 1	12%
synthesis	199 ± 33	4 ± 1	2%
sql	13 ± 1	3 ± 0	23%

Table 5: The ratio of the learning time (seconds) spent finding MUSPs.

5.3.2 *Q2. How Does* MUSPER *Compare to Other Approaches?*

Table 2 shows the predictive accuracies aggregated per domain of all the systems. MUSPER has higher predictive accuracy than METAGOL on every domain. MUSPER has equal or higher predictive accuracy than ALEPH on all but two domains: *trains*, where the difference is negligible, and *krk*.

The most comparable approach is DISCO. The results in Table 2 show that MUSPER has equal or higher predictive accuracy than DISCO on all the domains. Table 6 shows the learning times aggregated per domain for MUSPER and DISCO. The results show that MUSPER outperforms DISCO in 6/8 domains and that they are matched in the other two domains. A paired t-test confirms the significance of the difference ($p < 0.01$). MUSPER can reduce learning times by 97% compared to DISCO, such as on the *sql* domain.

The two main reasons for the performance difference between MUSPER and DISCO are that DISCO (i) only finds unsatisfiable rules from the BK alone and does not consider examples, and (ii) uses a predefined set of properties to find unsatisfiable rules. For instance, for the *sql-02* task, we want to learn an *out/2* relation using BK with *family/4* facts. For this task, DISCO does not learn any constraints from the BK. By contrast, MUSPER quickly learns many MUSPs, including:

$$\{\ \textit{out(A,B)} \leftarrow \textit{family(C,A,D,E)}\ \}$$

This rule is a MUSP because there is no positive example $out(A, B)$ where A is true because the second argument in a *family/4* fact in the BK. Because it only considers the BK and not the examples, DISCO cannot learn this MUSP. MUSPER also learns the MUSP:

$$\{\ \leftarrow \textit{family(D,B,C,E), family(C,D,E,A)}\ \}$$

DISCO cannot learn this MUSP because it is not one of its predefined properties. On this task, DISCO takes 1005s to terminate. By contrast, MUSPER learns an optimal solution and terminates in only 9s, a 97% improvement.

Overall, these results suggest that the answer to **Q2** is that MUSPER compares favourably to existing approaches, both in terms of predictive accuracies and learning times.

Domain	DISCO	MUSPER	Change
trains	14 ± 1	13 ± 1	**–7%**
zendo	40 ± 5	36 ± 4	**–10%**
imdb	250 ± 73	148 ± 39	**–40%**
iggp	583 ± 66	207 ± 34	**–64%**
krk	1219 ± 153	1214 ± 154	0%
graph	8 ± 2	8 ± 2	0%
synthesis	327 ± 38	199 ± 33	**–39%**
sql	505 ± 58	13 ± 1	**–97%**

Table 6: Learning times (seconds).

6 Conclusions and Limitations

We have introduced an approach that identifies *minimal unsatisfiable subprograms* (MUSPs). We have shown that pruning with MUSPs is optimally sound (Propositions 1 and 2) and effective (Propositions 3 and 4). We implemented our approach in MUSPER, which identifies the MUSPs of definite programs, including recursive programs. Our experimental results on many diverse domains show that our approach can drastically reduce learning times whilst maintaining high predictive accuracy, sometimes reducing learning times by 99%. In addition to the substantial empirical improvements, we think that a key benefit of this paper is its simplicity. We have shown that a well-studied idea in constraint satisfaction (finding minimal unsatisfiable cores) can drastically improve the performance of a machine learning algorithm on a range of diverse tasks.

Limitations

We consider definite (monotonic) programs. To learn non-monotonic programs we might need to consider methods than handle inconsistency in non-monotonic logics [51, 21].

To find MUSPs, we use what Dershowitz et al. [17] call a *folk algorithm* where we delete literals one by one. However, as Dershowitz et al. [17] point out, this folk algorithm does not scale well to large formulas. Although scalability is not an issue in our experiments, because the largest rule we learn has 10 literals, future work should address this limitation to scale to rules with hundreds of literals. In this case, more efficient methods, such as those described by Bendík and Černá [4], will be necessary to identify MUSPs.

Code and Data

A longer version of this paper with the appendices is available at https://arxiv.org/pdf/2401.16383.pdf. The experimental code and data are available at https://github.com/logic-and-learning-lab/ecai24-musper.

Acknowledgements

The authors are supported by the EPSRC fellowship (EP/V040340/1). For open access, the authors have applied a CC BY public copyright licence to any author-accepted manuscript version arising from this submission.

References

[1] H. Adé, B. Malfait, and L. De Raedt. RUTH: an ILP theory revision system. In *Methodologies for Intelligent Systems, 8th International Symposium, ISMIS 1994*, volume 869 of *Lecture Notes in Computer Science*, pages 336–345. Springer, 1994.

[2] M. Alviano, C. Dodaro, S. Fiorentino, A. Previti, and F. Ricca. ASP and subset minimality: Enumeration, cautious reasoning and muses. *Artif. Intell.*, 320:103931, 2023.

[3] A. Bembenek, M. Greenberg, and S. Chong. From SMT to ASP: solver-based approaches to solving datalog synthesis-as-rule-selection problems. *Proc. ACM Program. Lang.*, 7(POPL):185–217, 2023.

[4] J. Bendík and I. Černá. Must: minimal unsatisfiable subsets enumeration tool. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 135–152. Springer, 2020.

[5] H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artif. Intell.*, 101(1-2):285–297, 1998.

[6] N. Bramley, A. Rothe, J. Tenenbaum, F. Xu, and T. M. Gureckis. Grounding compositional hypothesis generation in specific instances. In *Proceedings of the 40th Annual Meeting of the Cognitive Science Society, CogSci 2018*, 2018.

[7] A. Bundy and B. Mitrovic. Reformation: A domain-independent algorithm for theory repair. Technical report, Technical report, University of Edinburgh, 2016.

[8] D. Corapi, A. Russo, and E. Lupu. Inductive logic programming in answer set programming. In *Inductive Logic Programming - 21st International Conference, ILP*, volume 7207 of *Lecture Notes in Computer Science*, pages 91–97. Springer, 2011.

[9] A. Cropper and C. Hocquette. Learning logic programs by discovering where not to search. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI*, pages 6289–6296. AAAI Press, 2023.

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

[11] A. Cropper and R. Morel. Predicate invention by learning from failures. *CoRR*, 2021.

[12] A. Cropper, S. Dumančić, and S. H. Muggleton. Turning 30: New ideas in inductive logic programming. In *IJCAI 2020*, pages 4833–4839, 2020.

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

[14] W. Dai and S. H. Muggleton. Abductive knowledge induction from raw data. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence 2021*, pages 1845–1851. ijcai.org, 2021.

[15] L. De Raedt and L. Dehaspe. Clausal discovery. *Mach. Learn.*, 26(2-3): 99–146, 1997.

[16] L. De Raedt, A. Dries, I. Thon, G. V. den Broeck, and M. Verbeke. Inducing probabilistic relational rules from probabilistic examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 1835–1843. AAAI Press, 2015.

[17] N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT 2006*, volume 4121, pages 36–41. Springer, 2006.

[18] S. Dumančić, T. Guns, W. Meert, and H. Blockeel. Learning relational representations with auto-encoding logic programs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pages 6081–6087. ijcai.org, 2019.

[19] R. Evans and E. Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018.

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

[21] J. K. Fichte, M. Hecher, and S. Szeider. Inconsistent cores for ASP: the perks and perils of non-monotonicity. In *AAAI 2023*, pages 6363–6371. AAAI Press, 2023.

[22] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *VLDB J.*, 24(6): 707–730, 2015.

[23] M. R. Genesereth and Y. Björnsson. The international general game playing competition. *AI Magazine*, 34(2):107–111, 2013.

[24] C. Glanois et al. Neuro-symbolic hierarchical rule induction. In *ICML 2022*, volume 162, pages 7583–7615. PMLR, 2022.

[25] C. Hocquette and S. H. Muggleton. Complete bottom-up predicate invention in meta-interpretive learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 2312–2318. ijcai.org, 2020.

[26] K. Inoue, T. Ribeiro, and C. Sakama. Learning from interpretation transition. *Mach. Learn.*, 94(1):51–79, 2014.

[27] T. Kaminski, T. Eiter, and K. Inoue. Meta-interpretive learning using hex-programs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pages 6186–6190. ijcai.org, 2019.

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

[29] N. Lavrac, B. Skrlj, and M. Robnik-Sikonja. Propositionalization and embeddings: two sides of the same coin. *Mach. Learn.*, 109(7):1465–1507, 2020.

[30] M. Law. Conflict-driven inductive logic programming. *Theory and Practice of Logic Programming*, 23(2):387–414, 2023.

[31] M. Law et al. Fastlas: scalable inductive logic programming incorporating domain-specific optimisation criteria. In *AAAI*, 2020.

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

[33] I. Lynce and J. P. M. Silva. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004*, 2004.

[34] M. Malec, T. Khot, J. G. Nagy, E. Blask, and S. Natarajan. Inductive logic programming meets relational databases: Efficient learning of markov logic networks. In *Inductive Logic Programming - 26th International Conference, ILP*, volume 10326 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2016.

[35] L. Mihalkova, T. N. Huynh, and R. J. Mooney. Mapping and revising markov logic networks for transfer learning. In *AAAI 2007*, pages 608–614, 2007.

[36] R. Morel and A. Cropper. Learning logic programs by explaining their failures. *Mach. Learn.*, 112(10):3917–3943, 2023.

[37] S. Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286, 1995.

[38] S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Mach. Learn.*, 100(1):49–73, 2015.

[39] A. Paes, G. Zaverucha, and V. S. Costa. On the use of stochastic local search techniques to revise first-order logic theories from examples. *Mach. Learn.*, 106(2):197–241, 2017.

[40] S. Patsantzis and S. H. Muggleton. Top program construction and reduction for polynomial time meta-interpretive learning. *Mach. Learn.*, 110(4):755–778, 2021.

[41] T. Piriyakulkij and K. Ellis. Doing experiments and revising rules with natural language and probabilistic reasoning. *CoRR*, abs/2402.06025, 2024.

[42] G. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, August 1971.

[43] S. J. Purgal, D. M. Cerna, and C. Kaliszyk. Learning higher-order logic programs from failures. In *IJCAI 2022*, pages 2726–2733. ijcai.org, 2022.

[44] M. Raghothaman, J. Mendelson, D. Zhao, M. Naik, and B. Scholz. Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, 4(POPL):62:1–62:27, 2020.

[45] O. Ray. Nonmonotonic abductive inductive learning. *J. Applied Logic*, 7 (3):329–340, 2009.

[46] B. L. Richards and R. J. Mooney. Automated refinement of first-order horn-clause domain theories. *Mach. Learn.*, 19(2):95–131, 1995.

[47] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983. ISBN 0262192187.

[48] X. Si, W. Lee, R. Zhang, A. Albarghouthi, P. Koutris, and M. Naik. Syntax-guided synthesis of datalog programs. In *ESE 2018*, pages 515–527. ACM, 2018.

[49] X. Si, M. Raghothaman, K. Heo, and M. Naik. Synthesizing datalog programs using numerical relaxation. In *IJCAI 2019*, pages 6117–6124, 2019.

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

[51] M. Ulbricht, M. Thimm, and G. Brewka. Handling and measuring inconsistency in non-monotonic logics. *Artif. Intell.*, 286:103344, 2020.

[52] S. Wrobel. Concept formation during interactive theory revision. *Mach. Learn.*, 14(1):169–191, 1994.

[53] Q. Zeng, J. M. Patel, and D. Page. Quickfoil: Scalable inductive logic programming. *Proc. VLDB Endow.*, 8(3):197–208, 2014.