ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230309

Learning Logic Programs by Combining Programs

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

University of Oxford

Abstract. The goal of inductive logic programming is to induce a logic program (a set of logical rules) that generalises training examples. Inducing programs with many rules and literals is a major challenge. To tackle this challenge, we introduce an approach where we learn small *non-separable* programs and combine them. We implement our approach in a constraint-driven ILP system. Our approach can learn optimal and recursive programs and perform predicate invention. Our experiments on multiple domains, including game playing and program synthesis, show that our approach can drastically outperform existing approaches in terms of predictive accuracies and learning times, sometimes reducing learning times from over an hour to a few seconds.

1 Introduction

The goal of inductive logic programming (ILP) [27, 8] is to induce a logic program (a set of logical rules) that generalises examples and background knowledge (BK). The challenge is to efficiently search a large hypothesis space (the set of all programs) for a *solution* (a program that correctly generalises the examples).

To tackle this challenge, divide-and-conquer approaches [4] divide the examples into subsets and search for a program for each subset. Separate-and-conquer approaches [28] search for a rule that covers (generalises) a subset of the examples, separate these examples, and then search for more rules to cover the remaining examples. Both approaches can learn programs with many rules and literals. However, as they only learn from a subset of the examples, they cannot perform predicate invention and struggle to learn recursive and optimal programs so tend to overfit.

To overcome these limitations, recent approaches [24, 22, 10, 14] use *meta-level* search [8] to learn optimal and recursive programs. However, most recent approaches struggle to learn programs with many rules in a program, many literals in a rule, or both. For instance, ASPAL [6] precomputes every possible rule allowed in a program and uses an answer set solver to find a subset that covers the examples. However, this now widely adopted precomputation approach [24, 22, 37, 34] does not scale to rules with more than a few literals. Likewise, many recent approaches struggle to learn programs with more than a few rules [13, 11, 10, 33, 19].

In this paper, our goal is to overcome the scalability limitations of recent approaches yet maintain the ability to learn recursive and optimal programs and perform predicate invention. The key idea is to first learn small *non-separable* programs that cover some of the examples and then search for a combination (a union) of these programs that covers all the examples.

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

Non-separable programs can be seen as building blocks for larger programs. A program h is *separable* when (i) it has at least two rules, and (ii) no predicate symbol in the head of a rule in h also appears in the body of a rule in h. A program is *non-separable* when it is not separable. For instance, consider the program p_1:

$$p_1 = \left\{ \begin{array}{l} \textit{happy(A)} \leftarrow \textit{rich(A)} \\ \textit{happy(A)} \leftarrow \textit{friend(A,B), famous(B)} \\ \textit{happy(A)} \leftarrow \textit{married(A,B), beautiful(B)} \end{array} \right\}$$

This program has three rules which can be learned separately and combined. In other words, the union of the logical consequences of each rule is equivalent to the logical consequences of p_1. Therefore, p_1 is separable. By contrast, consider the program p_2:

$$p_2 = \left\{ \begin{array}{l} \textit{happy(A)} \leftarrow \textit{rich(A)} \\ \textit{happy(A)} \leftarrow \textit{married(A,B), happy(B)} \end{array} \right\}$$

This program has two rules that cannot be learned separately because the second recursive rule depends on the first rule. In other words, the union of the logical consequences of each rule is not equivalent to the consequences of p_2. Therefore, p_2 is non-separable.

To explore this idea, we build on *learning from failures* (LFF) [10]. LFF frames the learning 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. For instance, POPPER, a LFF learner, uses a *generate*, *test*, and *constrain* loop to generate programs and test them on the examples. If a program is not a solution, POPPER builds constraints to explain why and uses these constraints to restrict future program generation. We use LFF to explore our idea because it supports learning optimal and recursive programs from infinite domains. Moreover, it is easily adaptable because of its declarative constraint-driven approach. We build on LFF by (i) only generating non-separable programs in the generate stage, and (ii) adding a combine stage to search for combinations of non-separable programs.

Motivating Example

We illustrate our approach with a simple example. Suppose we want to learn a program that generalises the following positive (E^+) and negative (E^-) examples of lists of numbers:

E^+ = *{f([1,3,5,7]), f([6,9,4,4]), f([21,22,23,24])}*
E^- = *{f([2,3,1]), f([9,3,2]), f([21,22,1])}*

For instance, we might want to learn a program such as:

$$h_0 = \left\{ \begin{array}{l} \textit{f(A)} \leftarrow \textit{head(A,7)} \\ \textit{f(A)} \leftarrow \textit{head(A,4), tail(A,B), head(B,4)} \\ \textit{f(A)} \leftarrow \textit{head(A,23), tail(A,B), head(B,24)} \\ \textit{f(A)} \leftarrow \textit{tail(A,B), f(B)} \end{array} \right\}$$

This program says that the relation *f(A)* holds for a list A if A contains the sequence [7] or [4,4] or [23,24]. The last recursive rule is important because it allows the program to generalise to lists of arbitrary length.

To find a program that generalises the examples, we use a generate, test, *combine*, and constrain loop. In the generate stage, we generate non-separable programs of increasing size (with one literal, two literals, etc), such as:

$$h_1 = \{\ f(A) \leftarrow head(A,1)\ \}$$
$$h_2 = \{\ f(A) \leftarrow head(A,2)\ \}$$
$$h_3 = \{\ f(A) \leftarrow head(A,3)\ \}$$

In the test stage, we test programs on the examples. If a program covers a negative example then we build a *generalisation* constraint to prune more general programs from the hypothesis space, as they will also cover the negative example. For instance, as h_2 covers the first negative example (*f([2,3,1])* we prune h_2 and its generalisations, such as h_4:

$$h_4 = \left\{ \begin{array}{l} f(A) \leftarrow head(A,2) \\ f(A) \leftarrow tail(A,B), f(B) \end{array} \right\}$$

If a program covers no positive examples then we build a *specialisation* constraint to prune more specific programs from the hypothesis space, as they will also not cover any positive examples. For instance, as we do not have a positive example where the first element is 3 we prune h_3 and its specialisations, such as h_5:

$$h_5 = \{\ f(A) \leftarrow head(A,3), tail(A,B), head(B,9)\ \}$$

If a program covers *at least one* positive and no negative examples we add it to a set of *promising* programs. For instance, when generating programs of size five, suppose we generate the recursive program:

$$h_6 = \left\{ \begin{array}{l} f(A) \leftarrow head(A,7) \\ f(A) \leftarrow tail(A,B), f(B) \end{array} \right\}$$

As h_6 covers at least one positive example (*f([1,3,5,7])*) and no negative examples we deem it a promising program.

In our novel combine stage, we then search for a combination of promising programs that covers all the positive examples and is minimal in size. We formulate this combinational problem as an answer set programming (ASP) problem [16], for which there are highly performance solvers, such as Clingo [17]. If we cannot find a combination, we go to the constrain stage where we use any discovered constraints to generate a new program. If we find a combination, we deem it the best solution so far. For instance, suppose that after considering programs of size seven we see the programs:

$$h_7 = \left\{ \begin{array}{l} f(A) \leftarrow head(A,4), tail(A,B), head(B,4) \\ f(A) \leftarrow tail(A,B), f(B) \end{array} \right\}$$
$$h_8 = \left\{ \begin{array}{l} f(A) \leftarrow head(A,23), tail(A,B), head(B,24) \\ f(A) \leftarrow tail(A,B), f(B) \end{array} \right\}$$

Then the combination of $h_6 \cup h_7 \cup h_8$ is h_0, the solution we want to learn. Crucially, we have learned a program with 4 rules and 13 literals by only generating programs with at most 2 rules and 7 literals. As the search complexity of ILP approaches is usually exponential in the size of the program to be learned, this reduction can substantially improve learning performance.

At this point, we have not proven that the combination is optimal in terms of program size. In other words, we have not proven that there is no smaller solution. Therefore, to learn an optimal solution, we continue to the constrain stage and add a constraint on the maximum program size (at most 12 literals) in future iterations. This constraint prohibits (i) any program with more than 12 literals from being generated in the generate stage, and (ii) any combination of promising programs with more than 12 literals from being found in the combine stage. We repeat this loop until we prove the optimality of a solution.

Novelty, impact, and contributions. The main novelty of this paper is the idea of *learning small non-separable programs that cover some of the examples and combining these programs to learn large programs with many rules and literals*. We expand on this novelty in Section 2. The impact, which our experiments conclusively show on many diverse domains, is vastly improved learning performance, both in terms of predictive accuracies and learning times, sometimes reducing learning times from over one hour to a few seconds. Moreover, as the idea connects many areas of AI , including program synthesis, constraint satisfaction, and logic programming, the idea should interest a broad audience.

Overall, we make the following contributions:

- We introduce a generate, test, combine, and constrain ILP approach.
- We implement our idea in COMBO, a new system that learns optimal and recursive programs and supports predicate invention. We prove that COMBO always returns an optimal solution if one exists.
- We experimentally show on many diverse domains, including game playing and program synthesis, that our approach can substantially outperform other approaches, especially in terms of learning times.

2 Related Work

Rule mining. ILP is a form of rule mining. A notable rule mining approach is AMIE+ [15]. Comparing COMBO with AMIE+ is difficult. AMIE+ adopts an open-world assumption. By contrast, COMBO adopts the closed-world assumption. Moreover, COMBO can learn programs with relations of arity greater than two, which AMIE+ cannot, i.e. AMIE+ can only use unary and binary relations. This difference is important as AMIE+ cannot be used on most of the datasets in our experiments. For instance, all the IGGP tasks [9] and many of the program synthesis tasks use relations of arity greater than 2, such *next_cell/3*, *true_cell/3* and *does_jump/4* in the *iggp-coins* task. Likewise, many of the program synthesis tasks use *append/3* or *sum/3*. Finally, AMIE+ requires facts as input, which can be difficult to provide, especially when learning from infinite domains such as the program synthesis domain. By contrast, COMBO takes as input a definite program as BK.

Classic ILP. TILDE [4] is a divide-and-conquer approach. PROGOL [28] is a separate-and-conquer approach that has inspired many other approaches [35, 2, 36], notably ALEPH [38]. Although both approaches can learn programs with many rules and literals, they struggle to learn recursive and optimal programs and cannot perform predicate invention [39].

Scalability. Scalability can be on many dimensions. For instance, many systems, such as QuickFOIL [40], focus on scaling to handle millions of training examples and background facts. Scaling to millions of examples is not a goal of this work (although we show that COMBO can handle hundreds of thousands of examples). Instead, our goal is to scale to scale to large hypothesis spaces, which is difficult for many existing systems, notably rule selection approaches, outlined in the following paragraph.

Rule selection. Many systems formulate the ILP problem as a rule selection problem [6, 24, 13, 22]. These approaches precompute every possible rule in the hypothesis space and then search for a subset that covers the examples, frequently using ASP to perform the search. The major limitation of precomputation approaches is scalability in terms of (i) the size of rules, and (ii) the number of possible rules. As these approaches precompute every possible rule, they cannot scale to rules with more than a few body literals because the number of rules is exponential in the number of body literals. Similarly, as they perform a combinatorial search over every possible rule, they cannot scale to problems with many possible rules. For instance, PROSYNTH [34] and DIFFLOG [37] consider at most 1000 and 1267 candidate rules respectively. However, our simplest experiment (*trains1*) requires 31,860 candidate rules. Our *coins-goal* experiment requires approximately 10^{15} candidate rules, which is infeasible for these approaches. Moreover, our approach differs in many ways. We do not precompute every possible rule, which allows us to learn rules with many body literals. In addition, we only search over promising programs (programs known to cover at least one positive and no negative example), which allows us to scale to problems with many possible rules.

LFF. Rather than precompute every possible rule, the key idea of POPPER is to discover constraints from smaller programs (potentially with multiple rules) to rule out larger programs. However, POPPER struggles to learn large programs with many rules and many literals because it tries to generate a single program that covers all the examples. We differ by (i) only generating non-separable programs in the generate step, and (ii) adding a combine step. DCC [7] combines classical divide-and-conquer search with modern constraint-driven ILP. DCC learns a program for each example separately. As these programs are likely to be overly specific, DCC iteratively tries to learn more general programs. To improve performance, DCC reuses knowledge between iterations. Our approach is completely different from DCC. DCC tries to generate a single program, potentially separable, that covers all the examples and has no combine stage. By contrast, COMBO never generates a separable program and searches for combinations of programs in the combine stage. HOPPER [33] extends POPPER to learn higher-order programs. Although our implementation only learns first-order programs, the approach should directly transfer to HOPPER.

3 Problem Setting

We now describe our problem setting. We assume familiarity with logic programming [25] and ASP [16] but have included a summary in the [41] appendix.

We use the LFF setting. A *hypothesis* is a set of definite clauses, i.e. we learn definite programs with the least Herbrand model semantics. We use the term *program* interchangeably with the term 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 meta-language that defines hypotheses. For instance, consider a meta-language formed of two literals *h_lit/3* and *b_lit/3* which represent *head* and *body* literals respectively. With this language, we can denote the rule *last(A,B)* ← *tail(A,C), head(C,B)* as the set of literals *{h_lit(0,last,(0,1)), b_lit(0,tail,(0,2)), b_lit(0,head,(2,1))}*. The first argument of each literal is the rule index, the second is the predicate symbol, and the third is the literal variables, where *0* represents *A*, *1* represents *B*, etc. 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. For instance, the rule *last(A,B)* ← *last(B,A)* violates the constraint ← *h_lit(0,last,(0,1)), b_lit(0,last,(1,0))*. We denote as $\mathcal{H}_C$ the subset of the hypothesis space $\mathcal{H}$ which does not violate any constraint in 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 an input tuple* $(E^+, E^-, B, \mathcal{H}, C)$*, 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 *(*$\forall e \in E^-,\ B \cup h \not\models e$*).*

If a hypothesis is not a solution then it is a *failure*. A hypothesis h 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 $\forall e \in E^+,\ h \cup B \not\models e$.

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

Definition 3 (Optimal solution) *Given an input tuple* $(E^+, E^-, B, \mathcal{H}, C)$*, 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,* $cost(h) \leq cost(h')$.

In this paper, our cost function is the number of literals in a hypothesis.

Constraints. The goal of a LFF learner is to learn hypothesis constraints from failed hypotheses. Cropper and Morel [10] introduce hypothesis constraints based on subsumption [32]. 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 definite theory t_1 subsumes a definite 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 definite theory t_1 is a *specialisation* of a definite theory t_2 if and only if $t_2 \preceq t_1$. A definite theory t_1 is a *generalisation* of a definite theory t_2 if and only if $t_1 \preceq t_2$. A *specialisation* constraint prunes specialisations of a hypothesis. A *generalisation* constraint prunes generalisations of a hypothesis.

4 Algorithm

We now describe our COMBO algorithm. To help explain our approach and delineate the novelty, we first describe POPPER.

POPPER. POPPER (Algorithm 1) solves the LFF problem. POPPER takes as input background knowledge (*bk*), positive (*pos*) and negative (*neg*) examples, and an upper bound (*max_size*) on hypothesis sizes. POPPER uses a generate, test, and constrain loop to find an optimal solution. POPPER starts with an ASP program $\mathcal{P}$ (hidden in the generate function). The models of $\mathcal{P}$ correspond to hypotheses (definite programs). In the generate stage (line 5), POPPER uses Clingo [17], an ASP system, to search for a model of $\mathcal{P}$. If there is no model, POPPER increments the hypothesis size (line 7) and loops again. If there is a model, POPPER converts it to a hypothesis h. In the test stage (line 9), POPPER uses Prolog to test h on the training examples. We use Prolog

because of its ability to handle lists and large, potentially infinite, domains. If h is a solution then POPPER returns it. If h is a failure then, in the constrain stage (line 12), POPPER builds hypothesis constraints (represented as ASP constraints) from the failure. POPPER adds these constraints to $\mathcal{P}$ to prune models, thus reducing the hypothesis space. For instance, if h is incomplete then POPPER builds a specialisation constraint to prune its specialisations. If h is inconsistent then POPPER builds a generalisation constraint to prune its generalisations. POPPER repeats this loop until it finds an optimal solution or there are no more hypotheses to test.

Algorithm 1 POPPER

```
1 def popper(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    cons += constrain(h, outcome)
13  return {}
```

4.1 COMBO

COMBO (Algorithm 2) builds on Algorithm 1 but differs by (i) only building non-separable programs in the generate stage, and (ii) adding a *combine* stage that tries to combine promising programs. We describe these novelties.

Algorithm 2 COMBO

```
1 def combo(bk, pos, neg, max_size):
2   cons = {}
3   promising = {}
4   best_solution = {}
5   size = 1
6   while size ≤ max_size:
7     h = generate_non_separable(cons, size)
8     if h == UNSAT:
9       size += 1
10      continue
11    outcome = test(pos, neg, bk, h)
12    if outcome == (PARTIAL_COMPLETE, CONSISTENT):
13      promising += h
14      combine_outcome = combine(promising, max_size, bk,
  neg)
15      if combine_outcome != NO_SOLUTION:
16        best_solution = combine_outcome
17        max_size = size(best_solution)-1
18    cons += constrain(h, outcome)
19  return best_solution
```

4.1.1 Generate

In the generate stage, COMBO only generates non-separable programs (line 7). A program h is *separable* when (i) it has at least two rules, and (ii) no predicate symbol in the head of a rule in h also appears in the body of a rule in h. A program is *non-separable* when it is not separable. For instance, COMBO cannot generate the following separable program:

$$p_1 = \left\{ \begin{array}{l} \textit{happy(A)} \leftarrow \textit{friend(A,B), famous(B)} \\ \textit{happy(A)} \leftarrow \textit{married(A,B), beautiful(B)} \end{array} \right\}$$

By only generating non-separable programs, we reduce the complexity of the generate stage. Specifically, rather than search over every possible program, COMBO only searches over non-separable programs, a vastly smaller space that notably excludes all programs with multiple rules unless they are recursive or use predicate invention. For instance, assume, for simplicity, that we have a problem with no recursion or predicate invention, that the rule space contains m rules, and that we allow at most k rules in a program. Then in the generate stage, POPPER searches over approximately m^k programs. By contrast, COMBO searches over only m programs.

To be clear, Algorithm 2 follows Algorithm 1 and uses an ASP solver to search for a constraint-consistent (non-separable) program. In other words, the *generate_non_separable* function in Algorithm 2 is the same as the *generate* function in Algorithm 1 except it additionally tells the ASP solver to ignore separable programs using the encoding described in Section B.1 in the [41] appendix.

4.1.2 Test and Constrain

If a program is partially complete (covers at least one positive example) and consistent, COMBO adds it to a set of promising programs (line 13). If a program is inconsistent, COMBO builds a generalisation constraint to prune its generalisations from the hypothesis space, the same as POPPER. If a program is partially complete and inconsistent then, unlike POPPER, COMBO does not build a specialisation constraint to prune its specialisations because we might want to specialise it. For instance, consider learning a program to determine whether someone is happy. Suppose COMBO generates the program:

$$\textit{happy(A)} \leftarrow \textit{rich(A)}$$

Suppose this program is partially complete and inconsistent. Then we still might want to specialise this program to:

$$\textit{happy(A)} \leftarrow \textit{rich(A), tall(A)}$$

This program might now be partially complete and consistent, so is a promising program. Therefore, COMBO only builds a specialisation constraint when a program is (i) totally incomplete, because none of its specialisations can be partially complete, or (ii) consistent, because there is no need to specialise a consistent program, as it could only cover fewer positive examples. We prove in the [41] appendix that these constraints do not prune optimal hypotheses.

4.1.3 Combine

In the novel combine stage (line 14), COMBO searches for a combination (a union) of promising programs that covers the positive examples and is minimal in size. We describe our combine algorithm in the next paragraph. If we cannot find a combination, we go to the constrain stage where we use any discovered constraints to generate a new program (line 18). If we find a combination, we deem it the best solution so far (line 16). To provably learn an optimal solution, we update the maximum program size (line 17) which prohibits any program with more than *max_size* literals from being generated in the

generate stage or any combination of promising programs with more than *max_size* literals from being considered in the combine stage. We repeat this loop until we prove the optimality of a solution.

Algorithm 3 shows the combine algorithm. To find a combination of promising programs, we follow ASPAL and formulate this combinational problem as an ASP problem. The function *build_encoding* builds the encoding (line 5). We briefly describe our encoding. The [41] appendix includes more details and an example encoding. We give each positive example a unique ID. For each rule in a promising program, we create a choice rule to indicate whether it should be in a solution. For each promising program, we add facts about its example coverage and size. We ask Clingo to find a model (a combination of rules) for the encoding (line 6) such that it (i) covers as many positive examples as possible, and (ii) is minimal in size. If there is no model, we return the best solution so far; otherwise, we convert the model to a program (line 6). Every combination program without recursion or predicate invention is guaranteed to be consistent (this result is an intermediate result in the proof of Theorem 1 in the [41] appendix). However, if a combination program has recursion or predicate invention then it could be inconsistent. In this case, we test the program on the negative examples to ensure consistency. If it is inconsistent, we add a constraint to the encoding (line 10) to eliminate this program and any generalistion of it from the combine encoding. We then loop again.

Algorithm 3 Combine

```
1 def combine(promising, max_size, bk, neg):
2   cons = {}
3   best_solution = NO_SOLUTION
4   while True:
5     encoding = build_encoding(promising, cons, max_size)
6     h = call_clingo(encoding)
7     if h == UNSAT:
8         break
9     if recursion_or_pi(h) and inconsistent(h, bk, neg):
10        cons += build_con(h)
11    else:
12        best_solution = h
13        break
14  return best_solution
```

A key advantage of our approach is that, whereas most rule selection approaches (Section 2), including ASPAL, assign a choice rule to every possible rule in the hypothesis space, we only do so to rules in promising programs. This difference is crucial to the performance of COMBO as it greatly reduces the complexity of the combinatorial problem. For instance, let m be the total number of possible rules and n be the number of promising programs. Then precomputation approaches search over 2^m programs whereas COMBO searches over 2^n programs. In practice n is vastly smaller than m so our combine stage is highly efficient. For instance, in our simplest experiment (*trains1*) there are 31,860 candidate rules so precomputation approaches search over 2^{31860} programs. By contrast, COMBO finds an optimal solution in four seconds by only searching over 10 promising programs, i.e. over 2^{10} programs.

Correctness. We prove the correctness of COMBO[1]:

Theorem 1 (Correctness) COMBO *returns an optimal solution if one exists.*

Due to space limitations, the proof is in the [41] appendix. At a high-level, to show this result, we show that (i) COMBO can generate and test every non-separable program, (ii) an optimal separable solution can be formed from a union (combination) of non-separable programs, and (iii) our constraints never prune optimal solutions.

5 Experiments

Our experiments aim to answer the question:

Q1 Can combining non-separable programs improve predictive accuracies and learning times?

To answer **Q1**, we compare the performance of COMBO against POPPER. As COMBO builds on POPPER, this comparison directly measures the impact of our new idea, i.e. it is the only difference between the systems.

To see whether COMBO is competitive against other approaches, our experiments aim to answer the question:

Q2 How does COMBO compare against other approaches?

To answer **Q2** we also compare COMBO against DCC, ALEPH, and METAGOL [30][2]. We use these systems because they can learn recursive definite programs. DCC, POPPER, and COMBO use identical biases so the comparison between them is fair. ALEPH uses a similar bias but has additional settings. We have tried to make a fair comparison but there will likely be different settings that improve the performance of ALEPH. The results for METAGOL are in the [41] appendix.

Methods

We measure predictive accuracy and learning time given a maximum learning time of 60 minutes. If a system does not terminate within the time limit, we take the best solution found by the system at that point. We repeat all the experiments 10 times and calculate the mean and standard error[3]. The error bars in the tables denote standard error. We round learning times over one second to the nearest second because the differences are sufficiently large that finer precision is unnecessary. We use a 3.8 GHz 8-Core Intel Core i7 with 32GB of ram. All the systems use a single CPU.

Domains

We use the following domains. The [41] 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 eastbound and westbound trains [23].

Chess. The task is to learn chess patterns in the king-rook-king (*krk*) endgame [20]. This dataset contains relations with arity greater than two, such as *distance/3* and *cell/4*.

Zendo. Zendo is a multiplayer game where players must discover a secret rule by building structures. Zendo is a challenging game that has attracted much interest in cognitive science [5].

[1] COMBO does not return every optimal solution. To do so, we can revise Algorithm 3 to ask the ASP solver to find and return all combinations at a certain size. Algorithm 2 would then maintain a set of all current best programs, rather than a single program.

[2] We also tried to compare COMBO against rule selection approaches. However, precomputing every possible rule is infeasible for our datasets. The [41] appendix contains more details.

[3] https://en.wikipedia.org/wiki/Standard_error

IMDB. This real-world dataset [26] contains relations between movies, actors, and directors. This dataset is frequently used to evaluate rule learning systems [12]. Note that this dataset has non-trivial numbers of training examples. For instance, the *imdb3* task has 121,801 training examples.

IGGP. The goal of *inductive general game playing* [9] (IGGP) is to induce rules to explain game traces from the general game playing competition [18]. This dataset is notoriously difficult for ILP systems: the currently best-performing system can only learn perfect solutions for 40% of the tasks. Moreover, although seemingly a toy problem, IGGP is representative of many real-world problems, such as inducing semantics of programming languages [3]. We use six games: *minimal decay (md)*, *rock - paper - scissors (rps)*, *buttons*, *attrition*, *centipede*, and *coins*. These tasks all require learning rules with relations of arity more than two, which is impossible for some rule learning approaches [15].

Graph problems. We use frequently used graph problems [13, 19]. All of these tasks require the ability to learn recursive programs.

Program synthesis. Inducing complex recursive programs is a difficult problem [29] and most ILP systems cannot learn recursive programs. We use a program synthesis dataset [10] augmented with two new tasks *contains* and *reverse*. The motivating example in the introduction describes the task *contains*. These tasks all require the ability to learn recursive programs and to learn from non-factual data.

5.1 Experimental Results

Q1. Can combining non-separable programs improve predictive accuracies and learning times?

Table 1 shows that COMBO (i) has equal or higher predictive accuracy than POPPER on all the tasks, and (ii) can improve predictive accuracy on many tasks. A McNemar's test confirms the significance of the difference at the $p < 0.01$ level. COMBO comprehensively outperforms POPPER when learning programs with many rules and literals. For instance, the solution for *buttons* (included in the [41] appendix) has 10 rules and 61 literals. For this problem, POPPER cannot learn a solution in an hour so has default accuracy. By contrast, COMBO learns an accurate and optimal solution in 23s.

Table 2 shows that COMBO has lower learning times than POPPER on all the tasks. A paired t-test confirms the significance of the difference at the $p < 0.01$ level. COMBO also outperforms POPPER when learning small programs. For instance, for *centipede* both systems learn identical solutions with 2 rules and 8 literals. However, whereas it takes POPPER 1102s (18 minutes) to learn a solution, COMBO learns one in 9s, a 99% reduction. COMBO also outperforms POPPER when learning recursive programs. For instance, for *reverse*, POPPER needs 1961s (30 minutes) to learn a recursive program with 8 literals, whereas COMBO only needs 44s, a 98% reduction.

To illustrate the efficiency of COMBO, consider the *coins-goal* task. For this task, there are approximately 10^{15} possible rules[4] and thus $\binom{10^{15}}{k}$ programs with k rules – which is why this task is infeasible for most rule selection approaches. Despite this large hypothesis space, COMBO finds an optimal solution in under two minutes. Moreover, the combine stage has a negligible contribution to the learning time. For instance, in one trial that took COMBO 97s to learn a solution, only 0.07s was spent in the combine stage.

[4] For *coins-goal*, 118 predicate symbols may appear in a rule, including symbols of arity 3 and 4. Assuming at most 6 variables in a rule, there are around 1200 possible body literals. Assuming at most 6 body literals, there are $\binom{1200}{6} \approx 10^{15}$ possible rules.

Task	COMBO	POPPER	DCC	ALEPH
trains1	**100 ± 0**	**100 ± 0**	**100 ± 0**	**100 ± 0**
trains2	98 ± 0	98 ± 0	98 ± 0	**100 ± 0**
trains3	**100 ± 0**	79 ± 0	**100 ± 0**	**100 ± 0**
trains4	**100 ± 0**	32 ± 0	**100 ± 0**	**100 ± 0**
zendo1	**97 ± 0**	**97 ± 0**	**97 ± 0**	90 ± 2
zendo2	**93 ± 2**	50 ± 0	81 ± 3	**93 ± 3**
zendo3	**95 ± 2**	50 ± 0	78 ± 3	**95 ± 2**
zendo4	**93 ± 1**	54 ± 4	88 ± 1	88 ± 1
imdb1	**100 ± 0**	**100 ± 0**	**100 ± 0**	**100 ± 0**
imdb2	**100 ± 0**	**100 ± 0**	**100 ± 0**	50 ± 0
imdb3	**100 ± 0**	50 ± 0	**100 ± 0**	50 ± 0
krk1	**98 ± 0**	**98 ± 0**	**98 ± 0**	97 ± 0
krk2	79 ± 4	50 ± 0	54 ± 4	**95 ± 0**
krk3	54 ± 0	50 ± 0	50 ± 0	**90 ± 4**
md	**100 ± 0**	37 ± 13	**100 ± 0**	94 ± 0
buttons	**100 ± 0**	19 ± 0	**100 ± 0**	96 ± 0
rps	**100 ± 0**	18 ± 0	**100 ± 0**	**100 ± 0**
coins	**100 ± 0**	17 ± 0	**100 ± 0**	17 ± 0
buttons-g	**100 ± 0**	50 ± 0	86 ± 1	**100 ± 0**
coins-g	**100 ± 0**	50 ± 0	90 ± 6	**100 ± 0**
attrition	**98 ± 0**	2 ± 0	2 ± 0	**98 ± 0**
centipede	**100 ± 0**	**100 ± 0**	81 ± 6	**100 ± 0**
adj_red	**100 ± 0**	**100 ± 0**	**100 ± 0**	50 ± 0
connected	**98 ± 0**	81 ± 7	82 ± 7	51 ± 0
cyclic	**89 ± 3**	80 ± 7	85 ± 5	50 ± 0
colouring	**98 ± 1**	**98 ± 1**	**98 ± 1**	50 ± 0
undirected	**100 ± 0**	**100 ± 0**	**100 ± 0**	50 ± 0
2children	**100 ± 0**	99 ± 0	**100 ± 0**	50 ± 0
dropk	**100 ± 0**	**100 ± 0**	**100 ± 0**	55 ± 4
droplast	**100 ± 0**	95 ± 5	**100 ± 0**	50 ± 0
evens	**100 ± 0**	**100 ± 0**	**100 ± 0**	50 ± 0
finddup	**99 ± 0**	98 ± 0	**99 ± 0**	50 ± 0
last	**100 ± 0**	**100 ± 0**	**100 ± 0**	55 ± 4
contains	**100 ± 0**	**100 ± 0**	99 ± 0	56 ± 2
len	**100 ± 0**	**100 ± 0**	**100 ± 0**	50 ± 0
reverse	**100 ± 0**	85 ± 7	**100 ± 0**	50 ± 0
sorted	**100 ± 0**	**100 ± 0**	**100 ± 0**	74 ± 2
sumlist	**100 ± 0**	**100 ± 0**	**100 ± 0**	50 ± 0

Table 1. Predictive accuracies.

Overall, these results strongly suggest that the answer to **Q1** is yes: learning non-separable programs and combining them can drastically improve learning performance.

Q2. How does COMBO *compare against other approaches?*

Table 1 shows that COMBO has equal or higher accuracy than DCC on all the tasks. A McNemar's test confirms the significance of the differences at the $p < 0.01$ level. COMBO has notably higher accuracy on the *iggp* and *zendo* tasks. Table 2 shows that COMBO has lower learning times than DCC on all the tasks. A paired t-test confirms the significance of the difference at the $p < 0.01$ level. COMBO finds solutions for all but one task within the time limit. By contrast, DCC times out on 12 tasks. For instance, for *buttons-g* it takes DCC over an hour to learn a solution with 86% accuracy. By contrast, COMBO finds a perfect solution in 3s, a 99% reduction. The [41] appendix includes the learning output from COMBO for this task.

Table 1 shows that ALEPH sometimes has higher accuracy than

Task	COMBO	POPPER	DCC	ALEPH
trains1	4 ± 0	5 ± 0	8 ± 1	$\mathbf{3 \pm 1}$
trains2	4 ± 0	82 ± 25	10 ± 1	$\mathbf{2 \pm 0}$
trains3	18 ± 1	*timeout*	*timeout*	$\mathbf{13 \pm 3}$
trains4	$\mathbf{16 \pm 1}$	*timeout*	*timeout*	136 ± 55
zendo1	3 ± 1	7 ± 1	7 ± 1	$\mathbf{1 \pm 0}$
zendo2	49 ± 5	*timeout*	3256 ± 345	$\mathbf{1 \pm 0}$
zendo3	55 ± 6	*timeout*	*timeout*	$\mathbf{1 \pm 0}$
zendo4	53 ± 11	3243 ± 359	2939 ± 444	$\mathbf{1 \pm 0}$
imdb1	$\mathbf{2 \pm 0}$	3 ± 0	3 ± 0	142 ± 41
imdb2	$\mathbf{3 \pm 0}$	11 ± 1	$\mathbf{3 \pm 0}$	*timeout*
imdb3	$\mathbf{547 \pm 46}$	875 ± 166	910 ± 320	*timeout*
krk1	28 ± 6	1358 ± 321	188 ± 53	$\mathbf{3 \pm 1}$
krk2	3459 ± 141	*timeout*	*timeout*	$\mathbf{11 \pm 4}$
krk3	*timeout*	*timeout*	*timeout*	$\mathbf{16 \pm 3}$
md	13 ± 1	3357 ± 196	*timeout*	$\mathbf{4 \pm 0}$
buttons	$\mathbf{23 \pm 3}$	*timeout*	*timeout*	99 ± 0
rps	87 ± 15	*timeout*	*timeout*	$\mathbf{20 \pm 0}$
coins	$\mathbf{490 \pm 35}$	*timeout*	*timeout*	*timeout*
buttons-g	$\mathbf{3 \pm 0}$	*timeout*	*timeout*	86 ± 0
coins-g	105 ± 6	*timeout*	*timeout*	$\mathbf{9 \pm 0}$
attrition	$\mathbf{26 \pm 1}$	*timeout*	*timeout*	678 ± 25
centipede	$\mathbf{9 \pm 0}$	1102 ± 136	2104 ± 501	12 ± 0
adj_red	$\mathbf{2 \pm 0}$	5 ± 0	6 ± 0	479 ± 349
connected	$\mathbf{5 \pm 1}$	112 ± 71	735 ± 478	435 ± 353
cyclic	$\mathbf{35 \pm 13}$	1321 ± 525	1192 ± 456	1120 ± 541
colouring	$\mathbf{2 \pm 0}$	6 ± 0	5 ± 0	2373 ± 518
undirected	$\mathbf{2 \pm 0}$	6 ± 0	6 ± 0	227 ± 109
2children	$\mathbf{2 \pm 0}$	7 ± 0	6 ± 0	986 ± 405
dropk	7 ± 3	17 ± 2	14 ± 2	$\mathbf{4 \pm 1}$
droplast	$\mathbf{3 \pm 0}$	372 ± 359	13 ± 1	763 ± 67
evens	3 ± 0	29 ± 3	25 ± 2	$\mathbf{2 \pm 0}$
finddup	11 ± 5	136 ± 14	149 ± 7	$\mathbf{0.8 \pm 0}$
last	$\mathbf{2 \pm 0}$	12 ± 1	11 ± 1	$\mathbf{2 \pm 0}$
contains	$\mathbf{17 \pm 0}$	299 ± 52	158 ± 48	64 ± 5
len	3 ± 0	52 ± 5	45 ± 2	$\mathbf{2 \pm 0}$
reverse	40 ± 5	1961 ± 401	1924 ± 300	$\mathbf{3 \pm 0}$
sorted	127 ± 78	111 ± 11	131 ± 10	$\mathbf{1 \pm 0}$
sumlist	4 ± 0	256 ± 27	221 ± 12	$\mathbf{0 \pm 0}$

Table 2. Learning times (seconds). A *timeout* entry means that the system did not terminate within 60 minutes.

COMBO, notably on the *krk* tasks[5]. However, COMBO comfortably outperforms ALEPH on most tasks, especially graphs and synthesis, which require recursion. COMBO notably outperforms ALEPH on the *iggp* tasks which do not require recursion. For instance, for *coins*, ALEPH cannot find a solution in an hour. By contrast, COMBO only needs 105s to learn a solution that has 100% accuracy. The [41] appendix includes a trace of the output from COMBO on this task.

Table 2 shows that ALEPH sometimes has lower learning times than COMBO. The reason is that, unlike COMBO, ALEPH is not guaranteed to find an optimal solution, nor tries to, so terminates quicker. For instance, for the *rps* task, ALEPH is faster (20s) than COMBO (87s). However, COMBO takes only 2s to find the same solution as Aleph but takes 87s to prove that it is optimal.

The results in the [41] appendix show that COMBO also outperforms METAGOL on all tasks.

[5] COMBO (and POPPER and DCC) struggles on this task because of an issue in the generate stage where the ASP solver struggles to find a model. We are unsure precisely why. To explore the issue, we ran Clingo with two threads, where each thread uses a different search heuristic. In this case, Clingo trivially finds a model, so we think that the default Clingo search heuristic just happens to struggle on the KRK tasks. To be clear, this issue is in the generate stage, not the combine stage.

Overall, these results strongly suggest that the answer to **Q2** is that COMBO compares favourably to existing approaches and can substantially improve learning performance, both in terms of predictive accuracies and learning times.

6 Conclusions and Limitations

We have introduced an approach that learns small non-separable programs that cover some training examples and then tries to combine them to learn programs with many rules and literals. We implemented this idea in COMBO, a new system that can learn optimal, recursive, and large programs and perform predicate invention. We showed that COMBO always returns an optimal solution if one exists. Our empirical results on many domains show that our approach can drastically improve predictive accuracies and reduce learning times compared to other approaches, sometimes reducing learning times from over 60 minutes to a few seconds. In other words, COMBO can learn accurate solutions for problems that other systems cannot. These substantial improvements should directly help many application areas, such as drug design [21], pathfinding [1], and learning higher-order programs [33].

Limitations and Future Work

Noise. In the combine stage, we search for a combination of programs that (i) covers all of the positive examples and none of the negative examples, and (ii) is minimal in size. Our current approach is, therefore, intolerant to noisy/misclassified examples. To address this limitation, we can relax condition (i) to instead find a combination that covers as many positive and as few negative examples.

Solvers. We formulate our combine stage as an ASP problem. We could, however, use any constraint optimisation approach, such as formulating it as a MaxSAT [31] problem. We therefore think this paper raises a challenge, especially to the constraint satisfaction and optimisation communities, of improving our approach by using different solvers and developing more efficient encodings.

Code, Data, and Appendices

A longer version of this paper with the appendices is available at https://arxiv.org/pdf/2206.01614.pdf. The experimental code and data are available at https://github.com/logic-and-learning-lab/ecai23-combo.

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] Forest Agostinelli, Rojina Panta, Vedant Khandelwal, Biplav Srivastava, Bharath Chandra Muppasani, Kausik Lakkaraju, and Dezhi Wu, 'Explainable pathfinding for inscrutable planners with inductive logic programming', in *ICAPS 2022 Workshop on Explainable AI Planning*, (2022).

[2] John Ahlgren and Shiu Yin Yuen, 'Efficient program synthesis using constraint satisfaction in inductive logic programming', *J. Machine Learning Res.*, (1), 3649–3682, (2013).

[3] Sándor Bartha and James Cheney, 'Towards meta-interpretive learning of programming language semantics', in *Inductive Logic Programming - 29th International Conference, ILP 2019, Plovdiv, Bulgaria, September 3-5, 2019, Proceedings*, pp. 16–25, (2019).

[4] Hendrik Blockeel and Luc De Raedt, 'Top-down induction of first-order logical decision trees', *Artif. Intell.*, (1-2), 285–297, (1998).

[5] Neil Bramley, Anselm Rothe, Josh Tenenbaum, Fei Xu, and Todd Gureckis, 'Grounding compositional hypothesis generation in specific instances', in *Proceedings of the 40th annual conference of the cognitive science society*, (2018).

[6] Domenico Corapi, Alessandra Russo, and Emil Lupu, 'Inductive logic programming in answer set programming', in *Inductive Logic Programming - 21st International Conference*, pp. 91–97, (2011).

[7] Andrew Cropper, 'Learning logic programs though divide, constrain, and conquer', in *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, pp. 6446–6453. AAAI Press, (2022).

[8] Andrew Cropper and Sebastijan Dumancic, 'Inductive logic programming at 30: A new introduction', *J. Artif. Intell. Res.*, **74**, 765–850, (2022).

[9] Andrew Cropper, Richard Evans, and Mark Law, 'Inductive general game playing', *Mach. Learn.*, (7), 1393–1434, (2020).

[10] Andrew Cropper and Rolf Morel, 'Learning programs by learning from failures', *Mach. Learn.*, (4), 801–856, (2021).

[11] Wang-Zhou Dai and Stephen H. Muggleton, 'Abductive knowledge induction from raw data', in *IJCAI 2021*, (2021).

[12] Sebastijan Dumančić, Tias Guns, Wannes Meert, and Hendrik Blockeel, 'Learning relational representations with auto-encoding logic programs', in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pp. 6081–6087, (2019).

[13] Richard Evans and Edward Grefenstette, 'Learning explanatory rules from noisy data', *J. Artif. Intell. Res.*, 1–64, (2018).

[14] Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek J. Sergot, 'Making sense of sensory input', *Artif. Intell.*, 103438, (2021).

[15] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek, 'Fast rule mining in ontological knowledge bases with AMIE+', *VLDB J.*, **24**(6), 707–730, (2015).

[16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub, *Answer Set Solving in Practice*, 2012.

[17] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub, 'Clingo = ASP + control: Preliminary report', *CoRR*, (2014).

[18] Michael R. Genesereth and Yngvi Björnsson, 'The international general game playing competition', *AI Magazine*, (2), 107–111, (2013).

[19] Claire Glanois, Zhaohui Jiang, Xuening Feng, Paul Weng, Matthieu Zimmer, Dong Li, Wulong Liu, and Jianye Hao, 'Neuro-symbolic hierarchical rule induction', in *International Conference on Machine Learning, ICML 2022*, volume 162, pp. 7583–7615. PMLR, (2022).

[20] Céline Hocquette and Stephen 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*, pp. 2312–2318, (2020).

[21] Rama Kaalia, Ashwin Srinivasan, Amit Kumar, and Indira Ghosh, 'Ilp-assisted de novo drug design', *Mach. Learn.*, (3), 309–341, (2016).

[22] Tobias Kaminski, Thomas Eiter, and Katsumi Inoue, 'Meta-interpretive learning using hex-programs', in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pp. 6186–6190, (2019).

[23] J. Larson and Ryszard S. Michalski, 'Inductive inference of VL decision rules', *SIGART Newsletter*, 38–44, (1977).

[24] Mark Law, Alessandra Russo, and Krysia Broda, 'Inductive learning of answer set programs', in *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014*, pp. 311–325, (2014).

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

[26] Lilyana Mihalkova, Tuyen Huynh, and Raymond J Mooney, 'Mapping and revising markov logic networks for transfer learning', in *Aaai*, volume 7, pp. 608–614, (2007).

[27] Stephen Muggleton, 'Inductive logic programming', *New Generation Computing*, (4), 295–318, (1991).

[28] Stephen Muggleton, 'Inverse entailment and progol', *New Generation Comput.*, (3&4), 245–286, (1995).

[29] Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan, 'ILP turns 20 - biography and future challenges', *Mach. Learn.*, (1), 3–23, (2012).

[30] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad, 'Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited', *Mach. Learn.*, (1), 49–73, (2015).

[31] Nina Narodytska and Fahiem Bacchus, 'Maximum satisfiability using core-guided maxsat resolution', in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, eds., Carla E. Brodley and Peter Stone, pp. 2717–2723. AAAI Press, (2014).

[32] G.D. Plotkin, *Automatic Methods of Inductive Inference*, Ph.D. dissertation, Edinburgh University, August 1971.

[33] Stanislaw J. Purgal, David M. Cerna, and Cezary Kaliszyk, 'Learning higher-order logic programs from failures', in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, ed., Luc De Raedt, pp. 2726–2733. ijcai.org, (2022).

[34] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz, 'Provenance-guided synthesis of datalog programs', *Proc. ACM Program. Lang.*, (POPL), 62:1–62:27, (2020).

[35] Oliver Ray, 'Nonmonotonic abductive inductive learning', *J. Applied Logic*, (3), 329–340, (2009).

[36] Peter Schüller and Mishal Benz, 'Best-effort inductive logic programming via fine-grained cost-based hypothesis generation - the inspire system at the inductive logic programming competition', *Mach. Learn.*, (7), 1141–1169, (2018).

[37] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik, 'Synthesizing datalog programs using numerical relaxation', in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, pp. 6117–6124, (2019).

[38] Ashwin Srinivasan, 'The ALEPH manual', *Machine Learning at the Computing Laboratory, Oxford University*, (2001).

[39] Irene Stahl, 'The appropriateness of predicate invention as bias shift operation in ILP', *Mach. Learn.*, (1-2), 95–117, (1995).

[40] Qiang Zeng, Jignesh M. Patel, and David Page, 'Quickfoil: Scalable inductive logic programming', *Proc. VLDB Endow.*, (3), 197–208, (2014).

[41] Andrew Cropper and Céline Hocquette, 'Learning logic programs by combining programs', arXiv preprint arXiv:2206.01614, (2022).