Machine learning manuscript No.
(will be inserted by the editor)

Learning efficient logic programs

Andrew Cropper · Stephen H. Muggleton

the date of receipt and acceptance should be inserted later

Abstract When machine learning programs from data, we ideally want to learn efficient rather than inefficient programs. However, existing inductive logic programming (ILP) techniques cannot distinguish between the efficiencies of programs, such as permutation sort ($n!$) and merge sort $O(n\ log\ n)$. To address this limitation, we introduce Metaopt, an ILP system which iteratively learns lower cost logic programs, each time further restricting the hypothesis space. We prove that given sufficiently large numbers of examples, Metaopt converges on minimal cost programs, and our experiments show that in practice only small numbers of examples are needed. To learn minimal time-complexity programs, including non-deterministic programs, we introduce a cost function called *tree cost* which measures the size of the SLD-tree searched when a program is given a goal. Our experiments on programming puzzles, robot strategies, and real-world string transformation problems show that Metaopt learns minimal cost programs. To our knowledge, Metaopt is the first machine learning approach that, given sufficient numbers of training examples, is guaranteed to learn minimal cost logic programs, including minimal time-complexity programs.

1 Introduction

Suppose you want to machine learn a program from the following examples:

```
f([p,r,o,g,r,a,m],r).
f([i,n,d,u,c,t,i,o,n],i).
```

These examples are described as Prolog facts. The first argument is the input list. The second argument is the output, and is the first duplicate element in the input. Given these examples and the background predicates shown in Figure 1, Metagol [25,3,4], a

A. Cropper
Department of Computer Science, University of Oxford, UK
E-mail: andrew.cropper@cs.ox.ac.uk

S. H. Muggleton
Department of Computing, Imperial College London, London, UK
E-mail: s.muggleton@imperial.ac.uk

state-of-the-art inductive logic programming (ILP) system, learns the program shown in Figure 2a. This recursive program goes through the elements of the list checking whether the same element exists in the rest of the list, i.e. finds the first duplicate element. By contrast, consider the program shown in Figure 2b, which was learned by Metaopt, a new ILP system introduced in Section 4. This program first sorts the list and then goes through checking whether any adjacent elements are the same. Although larger, both in terms of clauses and literals, the program learned by Metaopt is more efficient ($O(n\ log\ n)$) than the program learned by Metagol ($O(n^2)$). This efficiency difference is shown in Figure 2c, which compares the running times of the two programs on randomly generated examples.

```
head([H|_],H).
tail([_|T],T).
element([X|_],X):-!.
element([_|T],X):-element(T,X).
mergesort(A,B):- ...
```

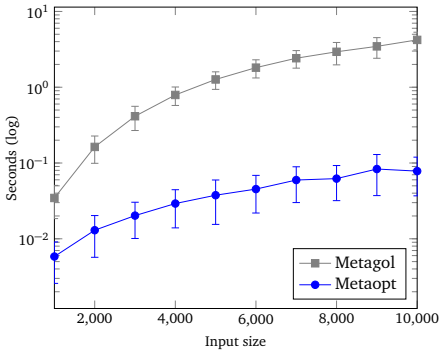
Fig. 1: Background knowledge for the *find duplicate* problem. The definition for *mergesort/2* is omitted for brevity.

```
f(A,B):-head(A,B),f_1(A,B).
f(A,B):-tail(A,C),f(C,B).
f_1(A,B):-tail(A,C),element(C,B).
```

(a)

```
f(A,B):-mergesort(A,C),f_1(C,B).
f_1(A,B):-head(A,B),f_2(A,B).
f_1(A,B):-tail(A,C),f_1(C,B).
f_2(A,B):-tail(A,C),head(C,B).
```

(b)



(c)

Fig. 2: Figure (a) shows an inefficient program learned by Metagol. Figure (b) shows an efficient program learned by Metaopt. The predicates `f1/2` and `f2/2` are invented. Figure (c) shows the running times of the two programs on randomly generated examples.

As this *find duplicate* scenario shows, when machine learning programs from examples, we should consider the efficiency of learned programs. However, existing ILP systems cannot distinguish between the efficiencies of programs, and typically rely on an Occamist bias to learn textually simple programs, such as those using the fewest literals [14] or fewest clauses [25].

A recent paper [2] attempted to address this issue by introducing Metagol$_O$, an ILP system based on meta-interpretive learning (MIL) [24,25,3]. Metagol$_O$ learns minimal

resource complexity robot strategies, described as dyadic logic programs, where resource complexity is the sum of the action costs required to execute a strategy.

In this paper, we introduce Metaopt, which generalises Metagol$_O$ by adding a general cost function into a meta-interpretive learner, where specific cost functions are provided as background knowledge. Metaopt uses a search procedure called *iterative descent*, introduced in [2] but generalised in this paper, to iteratively learn lower cost programs, each time further restricting the hypothesis space. We prove that given sufficiently large numbers of examples, Metaopt converges on minimal cost programs, and our experiments show that in practice only small numbers of examples are necessary. In contrast to Metagol$_O$, Metaopt is not restricted to dyadic logic programs, does not need designated input-output arguments, can handle negative examples, and considers backtracking steps when measuring program costs, crucial when learning non-deterministic logic programs. To learn minimal time-complexity programs, we introduce a cost function called *tree cost*, which measures the size of the SLD-tree searched when a program is given a goal. Our experiments on programming puzzles, robot strategies, and real-world string transformation problems show that Metaopt learns minimal tree cost programs.

Our specific contributions are as follows:

- We describe a general framework for learning minimal cost logic programs (Section 3).
- We extend MIL to support learning minimal cost logic programs (Section 3).
- To learn minimal time-complexity logic programs, we introduce a cost function called *tree cost* which is based on SLD-tree sizes (Section 3).
- We introduce Metaopt, a MIL implementation, and prove that it converges on minimal cost programs given sufficiently large numbers of examples (Section 4).
- We show through experimentation that Metaopt converges on minimal cost programs given small numbers of examples (Section 5).
- We demonstrate the generality of Metaopt by simulating Metagol$_O$ to learn efficient robot strategies (Section 5).
- We show that Metaopt learns more efficient programs than existing ILP systems on real-world string transformation problems (Section 5).

2 Related work

Universal search If nothing is known about a problem besides input/output examples, and assuming that the solution can be verified in polynomial time, Levin's universal search [15] is the asymptotically fastest way of finding a program to solve the problem. Levin search differs from our work because it returns the first (and smallest) program that solves a problem, which is not necessarily the most efficient program. By contrast, after finding a program, our approach, Metaopt, continues to search for more efficient programs using the cost of the previously found program to restrict the hypothesis space. In addition, for many problems, it is unlikely that the most efficient program can be encoded with a small number of bits, making Levin search impractical. By contrast, although our approach is less general, because it assumes background knowledge of the problem, it is more practical.

Deductive program synthesis In contrast to universal search methods, deductive program synthesis [17] approaches build programs from full specifications, where a specification

precisely states the requirements and behaviour of the desired program. Both Kant [10] and Zelle and Mooney [43] synthesise efficient programs from complete specifications. In [43] the authors take as input a program to sort lists and use an explanation-based learning [18] approach to speed-up the execution of the program by analysing program execution traces over training examples. A similar, yet distinct, approach is that of logic program transformation [29]. This approach starts with an initial program and successively applies transformation rules to programs to improve on previous versions. The aim is to end up with final program that has the same meaning as the initial one, but is preferably better in some way, such as being more compact or more efficient [29]. In contrast to these approaches, our approach induces programs from incomplete specifications in the form of input/output examples. In addition, the aforementioned approaches do not guarantee that the resulting program is optimal in terms of efficiency. By contrast, we prove that Metaopt finds the minimal cost program given sufficiently large numbers of examples (Theorem 1), and Experiment 1 shows that in practice only small numbers of examples are necessary.

Program induction Program induction, also known as inductive programming [8], refers to inducing programs from incomplete specifications, typically input/output examples. Early work includes Plotkin on least generalisation [30,31], Vere on induction algorithms for expressions in predicate calculus [39], and Summers on inducing Lisp programs [36]. Interest in program induction has grown recently, partially due to the success of mass-market tools, such as FlashFill [7]. Most forms of program induction are biased towards learning simple programs, typically those with minimal textual complexity, such as the number of clauses [25] or the number of literals [14]. This bias ignores the efficiency of hypothesised programs. In ILP, for instance, Golem [22] and Progol [21] can both learn sorting algorithms from examples, but when given background predicates *partition/3* and *append/3*, suitable for learning quick sort, both systems learn variants of insertion sort, because the definition is smaller and there is no bias for learning more efficient algorithms. By contrast, Metaopt is biased towards learning efficient programs.

AI planning In AI, planning typically involves deriving a sequence of actions to achieve a specific goal from an initial situation [33]. Planning research mostly focuses on efficiently learning plans [9]. However, we are often interested in plans that are optimal with respect to an objective function which measures the quality of a plan. A common objective function is the length of the plan [41], and existing systems can learn optimal plans based on this function [6].

Plan length alone is only one criterion. If executing actions is costly, we may prefer a plan which minimises the overall cost of the actions, e.g. to minimise the use of resources. The answer set programming literature has started to address learning optimal plans by incorporating action costs into the learning [6,42]. In contrast to these approaches, in Section 5.4, we use Metaopt to learn robot *strategies* [2], representing a (sometimes infinite) set of plans, which contain conditions and recursion.

Various machine learning approaches support constructing strategies, such as the SOAR architecture [13], reinforcement learning [37], and action learning in ILP [19, 28]. Relational markov decision processes [38] provide a general setting for reinforcement learning. Strategies can be viewed as a deterministic special case of markov decision processes (MDPs) [32]. Unlike these approaches, we learn recursive logic programs, including the use of predicate invention for automatic problem decomposition.

Efficient logic programs Kaplan [11] describes a method for estimating the average-case complexity of deterministic logic programs. However, in contrast to functional and imperative programs, logic programs can be non-deterministic, i.e. a logic program may return multiple solutions. Multiple solutions to a logic program are found by searching a SLD-tree for a SLD-refutation, and then backtracking to find other SLD-refutations. Debray et al. [5] introduce a semi-automatic method to estimate the worst-case time complexity of deterministic and non-deterministic logic programs. However, their approach requires meta-information, such as mode declarations and type information. In contrast to these approaches, we introduce a cost function which estimates the worst-case complexity of both deterministic and non-deterministic logic programs. Our approach does not require meta-information, and works by measuring the size of the SLD-tree searched to find a SLD-refutation of a goal. In addition, the aforementioned approaches do not consider how to machine learn efficient programs.

In Section 4, we introduce Metaopt, which is used in Experiments 1, 2, and 3 to learn minimal time-complexity programs by iteratively restricting the hypothesis space by bounding the number of resolutions allowed to find a hypothesis. Our approach is similar to one proposed by Blum and Blum [1] which was later adapted by Shapiro [34] who uses the notion of *h-easy* functions to limit the search for a hypothesis, where an atom A is *h-easy* with respect to a logic program P if there exists a derivation of A from P using at most h resolution steps. Shapiro's approach measures the number of resolutions in the derivation of A from P, and thus ignores backtracking steps. By contrast, our approach uses the notion of *tree cost* to measure the total number of resolutions required to find a SLD-refutation of a goal with respect to a program, i.e. tree cost includes backtracking steps.

Our approach of evaluating a hypothesis based on the SLD-tree size is similar the the idea of Muggleton et al. [23], who combined proof complexity (the number of choice points in a SLD-refutation) with other factors, such as hypothesis length and predictive accuracy, to measure the *significance* of a hypothesis. However, the authors only considered how to characterise the significance of a hypothesis. By contrast, we use a meta-interpretive learner to simultaneously induce and evaluate the time complexity of a hypothesis.

Metagol$_O$ To our knowledge, the only work on inducing efficient programs is the work of Cropper and Muggleton [2], who introduce an ILP system called Metagol$_O$ and show that it learns efficient robot strategies. A robot strategy is a dyadic logic program where the first argument of each predicate is the input and the second argument is the output. Each argument is a state description, represented as a list of Prolog atoms. Each predicate represents an action which modifies the state. The resource complexity of a strategy is maintained in the state as a monadic atom named *energy*. Each time a robot action is successfully executed, the resource complexity is increased by an amount specified by the user. Metagol$_O$ is not a general approach for learning efficient logic programs and has several limitations. Problems must be represented as dyadic programs, which is inconvenient and may lead to reduced learning performance because concepts may be less succinctly represented. Also, the user must specify predicate costs. For instance, if *mergesort/2* is part of the background knowledge, then the user must specify its cost. If no costs are given, Metagol$_O$ assumes uniform costs, and cannot, for instance, distinguish between *mergesort/2* and *tail/2*. Finally, Metagol$_O$ ignores failed actions and cannot accurately measure the time complexity of programs with backtracking. Our system, Metaopt, addresses all of these issues.

3 Framework

In this section, we describe the *cost minimisation problem*. We also describe MIL, which we extend to support learning minimal cost programs. We assume familiarity with logic programming [26].

3.1 Cost minimisation problem

We assume a language of examples $\mathscr{E}$, background knowledge $\mathscr{B}$, and hypotheses $\mathscr{H}$. We denote the Herbrand base of the language L as σ_L. We denote the power set of the set S as 2^S.

We first define a cost function that measures the cost of a program with respect to an atom:

Definition 1 (Program cost) A program cost function is of the form:

$$\Phi : \mathscr{H} \times \sigma_{\mathscr{B} \cup \mathscr{E}} \rightarrow \mathbf{N}$$

A program cost function forms part of the cost minimisation input:

Definition 2 (Cost minimisation input) The cost minimisation input is a triple (B, E, Φ) where:

- $B \subseteq \mathscr{B}$ is background knowledge
- $E = (E^+, E^-)$ is a pair where $E^+ \subseteq \mathscr{E}, E^- \subseteq \mathscr{E}$ are sets of atoms representing positive and negative examples respectively
- Φ is a program cost function

We measure the maximum (i.e. worst-case) cost of a program over a set of examples:

Definition 3 (Maximum cost) Let (B, E, Φ) be a cost minimisation input, where $E = (E^+, E^-)$. Then the maximum cost of a program $H \in \mathscr{H}$ is:

$$max\_cost(\Phi, H, (E^+, E^-)) = max_{e \in E^+ \cup E^-} \Phi(H, e)$$

We also define a function that measures the size of a logic program:

Definition 4 (Program size) The size $size(H)$ of the program H is the number of clauses in H.

We use the maximum cost and size of a program to define an efficiency ordering over programs:

Definition 5 (Efficiency ordering $\preceq_\Phi$) Let (B, E, Φ) be a cost minimisation input and $H_1, H_2 \in \mathscr{H}$. Then $H_1 \preceq_\Phi H_2$ iff either:

1. $max\_cost(\Phi, H_1, E) < max\_cost(\Phi, H_2, E)$
2. $max\_cost(\Phi, H_1, E) = max\_cost(\Phi, H_2, E)$ and $size(H_1) \leq size(H_2)$

This ordering priorities programs first by their maximum cost, then by their size. We use this ordering to define the cost minimisation problem. For convenience, we first define the version space [18], which contains only hypotheses consistent with the examples:

Definition 6 (Version space) The version space $\mathscr{V}_{B,E}$ of a cost minimisation input (B,E,Φ) contains the hypotheses consistent with E:

$$\mathscr{V}_{B,E} = \{H \in \mathscr{H} \mid B \cup H \models E^+, B \cup H \not\models E^-\}$$

We now define the cost minimisation problem:

Definition 7 (Cost minimisation problem) Given a cost minimisation input (B,E,Φ), the cost minimisation problem is to return a program $H \in \mathscr{V}_{B,E}$ such that $H \preceq_\Phi H'$ for all $H' \in \mathscr{V}_{B,E}$.

3.2 Meta-interpretive learning

We now extend MIL to support the cost minimisation problem. MIL is a form of ILP based on a Prolog meta-interpreter. The key difference between a MIL learner and a standard Prolog meta-interpreter is that whereas a standard Prolog meta-interpreter attempts to prove a goal by repeatedly fetching first-order clauses whose heads unify with a given goal, a MIL learner additionally attempts to prove a goal by fetching higher-order metarules (Figure 3), supplied as background knowledge, whose heads unify with the goal. The resulting meta-substitutions are saved and can be reused in later proofs. Following the proof of a set of goals, a logic program is formed by projecting the meta-substitutions onto their corresponding metarules.

Name	**Metarule**
Ident	$P(A,B) \leftarrow Q(A,B)$
Precon	$P(A,B) \leftarrow Q(A),R(A,B)$
Curry	$P(A,B) \leftarrow Q(A,B,F)$
Chain	$P(A,B) \leftarrow Q(A,C),R(C,B)$
Tailrec	$P(A,B) \leftarrow Q(A,C),P(C,B)$

Fig. 3: Example metarules. The letters P, Q, R, and F denote existentially quantified variables. The letters A, B, and C denote universally quantified variables.

A standard MIL input is defined as:

Definition 8 (MIL input) A MIL input is a pair (B,E) where:

- $B = B_C \cup M$ where B_C is a set of definite clauses and M is a set of metarules
- $E = (E^+, E^-)$ is a pair where E^+ and E^- are sets of atoms representing positive and negative examples respectively

A standard MIL learner is defined as:

Definition 9 (MIL learner) Given a MIL input (B,E), a MIL learner returns a program $H \in \mathscr{V}_{B,E}$.

We extend MIL to support the cost minimisation problem. We first extend the MIL input:

Definition 10 (Cost minimal MIL input) A cost minimal MIL input is a triple (B,E,Φ) where B and E are as in a standard MIL input and Φ is a program cost function.

We now define a cost minimal MIL learner:

Definition 11 (Cost minimal MIL learner) Given a cost minimal MIL input (B, E, Φ), a cost-minimal MIL learner returns a program $H \in \mathscr{V}_{B,E}$ such that $H \preceq_\Phi H'$ for all $H' \in \mathscr{V}_{B,E}$.

In Section 4, we introduce Metaopt, a MIL learner that solves the MIL cost minimisation problem.

3.3 Tree cost minimisation

The cost minimisation input includes a program cost function (Definition 1). We now introduce a cost function for learning minimal time-complexity logic programs. In computer science, time complexity refers to the time an algorithm needs to perform some computation. In logic programming, computation is formalised by means of SLD-resolution. Given a logic program H and a goal G, computation involves finding a SLD-refutation of $H \cup \{G\}$. A SLD-refutation is found by searching a SLD-tree, which contains all possible SLD-derivations, and thus all possible SLD-refutations. Prolog searches for SLD-refutations using a depth-first search [35]. We can therefore measure the runtime (time complexity) of a Prolog program as a function of the size of the SLD-tree that is being searched. For a positive example, we can measure the size of the leftmost branch of the SLD-tree in which the first SLD-refutation is found, i.e. the leftmost successful branch:

Definition 12 (Successful branch) Let H be a definite program, G be an initial goal, and T be a SLD-tree for $H \cup \{G\}$. Then a successful branch is a path between the root (G) and a leaf containing the empty clause.

We measure the size of the leftmost successful branch:

Definition 13 (Branch size) Let H be a definite program, G a goal, T a SLD-tree for $H \cup \{G\}$, and L be the leftmost successful branch of T. Then the branch size $branch\_size(H, G)$ is the number of resolutions prior to and including L in the depth-first enumeration of T.

For a negative example, we can measure the size of the finitely failed SLD-tree:

Definition 14 (Finitely failed tree) Let H be a definite program, G be an initial goal. Then a finitely failed SLD-tree for $H \cup \{G\}$ is one which is finite and contains no successful branches.

We measure the size of the finitely failed SLD-tree:

Definition 15 (Failed tree size) Let H be a definite program, G be an initial goal, and T be a finitely failed SLD-tree for $H \cup \{G\}$. Then the failed tree size $tree\_size(H, G)$ is the number of resolutions in the depth-first enumeration of T.

We now define our *tree cost* function:

Definition 16 (Tree cost) Let H be a definite program, G a goal, and T be a SLD-tree for $H \cup \{G\}$. Then the tree cost $tree\_cost(H, G)$ is:

$$tree\_cost(H,G) = \begin{cases} branch\_size(H,G) & \text{if } T \text{ has a successful branch} \\ tree\_size(H,G) & \text{if } T \text{ is finitely failed} \end{cases}$$

In Experiments 1, 2, and 3, Metaopt uses tree cost to learn minimal time-complexity programs.

4 Implementation

In this section, we introduce Metaopt, a MIL implementation that learns minimal cost logic programs. We also introduce two cost function implementations for learning minimal tree cost (Definition 16) and minimal resource complexity [2] programs. Finally, we describe Metagol$_O$ which is used as a comparator in the experiments in Section 5.

4.1 Metaopt

Metaopt extends Metagol [4], an existing MIL implementation, to support learning minimal cost logic programs. The two key extensions are (1) the addition of a general cost function into the meta-interpreter, and (2) the use of a procedure called iterative descent to search for lower cost programs. We describe these two extensions in turn.

4.1.1 Meta-interpreter

The key extension in Metaopt is the addition of a proof cost, denoted by the variables C_i, into the meta-interpreter[1]:

```
learn(Pos,Neg,Prog):-
    prove(Pos,[],Prog,0,_),
    \+ prove(Neg,Prog,Prog,0,_).
prove([],Prog,Prog,C,C).
prove([Atom|Atoms],Prog1,Prog2,C1,C2):-
    prove_aux(Atom,Prog1,Prog3,C1,C3),
    prove(Atoms,Prog3,Prog2,C3,C2).
prove_aux(Atom,Prog,Prog,C1,C2):-
    pos_program_cost(Atom,Cost),
    C2 is C1+Cost,
    get_max_cost(MaxCost),
    C2<MaxCost.
prove_aux(Atom,Prog1,Prog2,C1,C2):-
    member(sub(Name,Subs),Prog1),
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,Prog1,Prog2,C1,C2).
prove_aux(Atom,Prog1,Prog2,C1,C2):-
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,[sub(Name,Subs)|Prog1],Prog2,C1,C2).
```

The meta-interpreter works as follows. Given sets of atoms representing positive (`Pos`) and negative (`Neg`) examples, Metaopt tries to prove each positive atom in turn. Metaopt first tries to deductively prove an atom by calling `pos_program_cost/2`, which is defined as background knowledge. When an atom is proven this way, the cost of proving that atom is added to the overall proof cost. If the overall proof cost exceeds a bound (`MaxCost`), then the proof is terminated, as to ignore inefficient programs. This bound is determined by the iterative descent procedure, described in the next section. If Metaopt

[1] To aid readability, the Prolog code for the meta-interpreter is a slimmed down version of the one used in the experiments

cannot deductively prove an atom, it tries to unify the atom with the head of a metarule (`metarule(Name,Subs,(Atom :- Body))`) and to bind the existentially quantified variables in a metarule to symbols in the predicate signature. Metaopt saves the resulting meta-substitutions, which are eventually used to form a program. Metaopt then tries to prove the body of the metarule recursively through meta-interpretation. After proving all positive atoms, a logic program is formed by projecting the meta-substitutions onto their corresponding metarules. Metaopt checks the consistency of the learned program with the negative examples. If the program is inconsistent, then Metaopt backtracks to explore different branches of the SLD-tree.

4.1.2 Iterative descent

The Metaopt meta-interpreter is controlled by the iterative descent algorithm:

```
metaopt(Pos,Neg):-
    learn(Pos,Neg,Prog),
    max_program_cost(Prog,Pos,Neg,Cost),
    is_better(Cost),
    set_max_cost(Cost),
    set_best_program(Prog),
    false.
metaopt(_,_):-
    get_best_program(Prog),
    pprint(Prog).
max_program_cost(Prog,Pos,Neg,MaxCost):-
    assert_program(Prog),
    findall(C,(member(Atom,Pos),pos_program_cost(Atom,C)),PosCosts),
    findall(C,(member(Atom,Neg),neg_program_cost(Atom,C)),NegCosts),
    append(PosCosts,NegCosts,Costs),
    max_list(Costs,MaxCost),
    retract_program(Prog).
```

Iterative descent works as follows. Starting at iteration 1, Metaopt uses iterative deepening on the number of clauses to find a consistent program H_1 with the minimal program size (Definition 4). The program H_1 is the quickest to learn because the hypothesis space is exponential in the number of clauses [16,3]. Metaopt then calculates the cost of H_1 using the `max_program_cost/4` predicate, which measures the maximum cost of proving the positive examples and disproving the negative examples. This cost becomes the maximum cost for the next iteration for both the meta-interpreter and the iterative descent algorithm. In iteration $i > 1$, Metaopt searches for a program H_i, again with minimal program size, but ensuring that the cost of H_i is less than the cost of H_{i-1}. Iterative descent continues until it cannot find a lower cost program.

We now prove that Metaopt solves the cost minimisation problem (Definition 7), i.e. that Metaopt converges on minimal cost programs given sufficiently large numbers of examples:

Theorem 1 (Metaopt convergence) *Assume E consists of m examples drawn from an enumeration of the infinite example space E'. Without loss of generality consider the hypothesis space formed of two programs H_1 and H_2 such that $H_1 \preceq_\Phi H_2$ for an arbitrary cost function Φ. Then for a sufficiently large value m, Metaopt will return H_1 in preference to H_2.*

Proof Assume false, which, because of Definition 5, implies that one of the following conditions must hold:

1. $max\_cost(\Phi, H_1, E) > max\_cost(\Phi, H_2, E)$
2. $max\_cost(\Phi, H_1, E) = max\_cost(\Phi, H_2, E)$ and $size(H_1) > size(H_2)$

We consider these two cases in turn:

Case 1 With sufficiently large m there will exist an example e such that $\Phi(H_1, e) < \Phi(H_2, e)$ and $\Phi(H_2, e) > \Phi(H_2, e')$ for all other e' in E and $\Phi(H_1, e) > \Phi(H_1, e')$ for all other e' in E. In this case $max\_cost(\Phi, H_1, E) < max\_cost(\Phi, H_2, E)$ and Metaopt returns H_1 which contradicts the assumption, so we discard this case.

Case 2 Metaopt performs iterative deepening search (IDS) on the number of clauses. From the optimality of IDS, Metaopt returns H_1 which contradicts the assumption, so we discard this case.

These two cases are exhaustive, thus the proof is complete.

4.2 Program costs

Metaopt assumes a program cost function (Definition 1) as background knowledge. We now describe two cost function implementations.

4.2.1 Tree cost

Figure 4 shows the implementation of the tree cost functions (Definition 16) for positive and negative examples. This implementation uses an inbuilt feature of SWI-Prolog [40] to measure the number of logical inferences needed to prove an atom, where an inference is defined as a call or redo on a predicate[2]. This approach measures backtracking steps, costs associated with non-dyadic predicates, and costs associated with trying to prove negative examples, none of which are supported by Metagol$_O$. In Experiments 1, 2, and 3, Metaopt uses this implementation to learn minimal tree cost programs, and thus minimal time-complexity programs.

4.2.2 Resource complexity

In Experiment 4, we reproduce an experiment from [2] to show that Metaopt can learn minimal resource complexity robot strategies and thus subsumes Metagol$_O$. As explained in Section 2, Metagol$_O$ maintains the resource complexity of a strategy in the state description as a Prolog fact called *energy*. Figure 5 shows the implementation of a function which accesses this value so that it can be used by Metaopt.

4.3 Metagol$_O$

In Experiments 2 and 3 we compare Metaopt with Metagol$_O$. However, we cannot use the implementation from [2] because Metagol$_O$ requires that problems be represented

[2] http://www.swi-prolog.org/pldoc/man?predicate=statistics/2

```
pos_program_cost(Atom,Cost):-
    statistics(inferences,I1),
    call(Atom),
    statistics(inferences,I2),
    Cost is I2-I1.

neg_program_cost(Atom,Cost):-
    statistics(inferences,I1),
    (call(Atom) -> true; true),
    statistics(inferences,I2),
    % subtract one to account for the
    % cost of the ifthen statement
    Cost is I2-I1-1.
```

Fig. 4: Implementation of a program cost function to measure tree cost

```
pos_program_cost(Atom,Cost):-
    Atom=..[P,A,B],
    member(energy(E1),A),
    call(Atom),
    member(energy(E2),B),
    Cost is E2-E1.
neg_program_cost(_,_):-
    false.
```

Fig. 5: Implementation of a program cost function to measure resource complexity

as dyadic robot strategies (as detailed in Section 2). Therefore, we simulate Metagol$_O$ in Metaopt by defining a program cost predicate in which all dyadic predicates have a uniform cost of 1, which is the assumption in Metagol$_O$, and non-dyadic predicates have a cost of 0, because Metagol$_O$ does not take these into account when calculating resource complexity.

5 Experiments

We now describe four experiments[3] which test whether Metaopt learns minimal cost programs. We also compare Metaopt with Metagol and Metagol$_O$, which minimise textual complexity and resource complexity respectively.

5.1 Experiment 1: convergence on minimal cost programs

This experiment revisits the *find duplicate* problem from Section 1, where we are trying to learn a program to find a duplicate in a list. The aim is to test the claim that Metaopt

[3] All code and experimental data used in the experiments are available at https://github.com/andrewcropper/mlj18-metaopt

converges on minimal cost programs given sufficient examples (Theorem 1). In particular, we want to see how many examples are required in practice to converge on a minimal cost program. We test the null hypothesis:

Null hypothesis 1 Metaopt cannot learn minimal cost programs without very large numbers of examples.

This experiment focuses on learning minimal tree cost programs, and thus minimal time-complexity programs.

Materials To refute null hypothesis 1, we must identify a minimal tree cost program in the hypothesis space, otherwise our hypothesis would be untestable. We provide Metaopt with background knowledge containing the *ident*, *chain*, and *tailrec* metarules (Figure 3) and four predicates: *mergesort/2*, *tail/2*, *head/2*, and *element/2* (Figure 1). Given this background knowledge, the minimal tree cost program in the hypothesis space is shown in Figure 2b[4]. This minimal cost program has the following tree cost:

Proposition 1 ***Find duplicate minimal cost program*** *Let n be the list length. Then the tree cost of the minimal tree cost program is $O(n\ log\ n)$.*

Sketch proof 1 The minimal tree cost program involves first sorting the list and then passing through the list checking whether any two adjacent elements are the same. Thus the overall cost is $O(n\ log\ n)$.

We generate a positive training example as follows:

1. Select a random integer k from the interval $[5, 100]$ to represent the size of the input
2. Select a random integer j from the interval $[1, k]$ to represent the duplicate element
3. Append j to the sequence $1 \dots k$ and randomly shuffle the resulting list to form s'
4. Form the atom $f(s', j)$

We generate a negative training example as follows:

1. Select a random integer k from the interval $[5, 100]$ to represent the size of the input
2. Select a random integer j from the interval $[1, k]$ to represent the false duplicate element
3. Randomly shuffle the sequence $1 \dots k$ to form s'
4. Form the atom $f(s', j)$

We generate testing examples using the same procedures but for a fixed input size $k = 1000$.

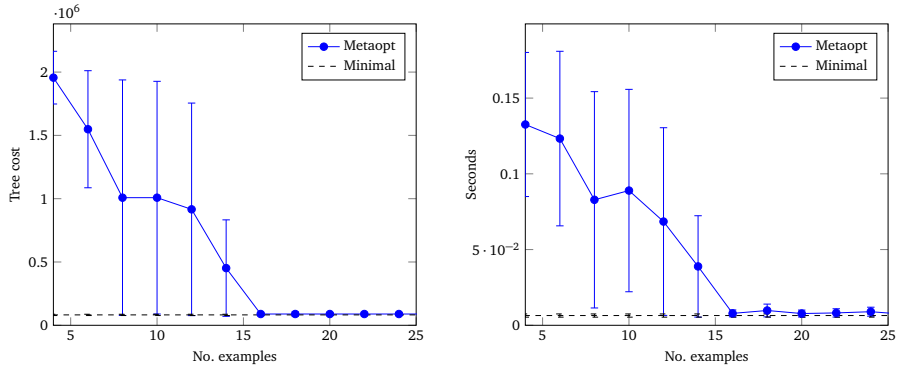
Method Our experimental method is as follows. For each m in the set $\{4, 6, 8, \dots, 40\}$:

1. Generate m training examples, half positive and half negative
2. Generate 2000 testing examples, half positive and half negative
3. Learn a program p using the training examples with a timeout of 10 minutes.
4. Measure the tree cost and running time of p over the testing examples

We measure median tree costs and running times over 20 repetitions.

[4] One could find the duplicate in time O(n) using a hash table but this program is not in the hypothesis space, so could not be found by Metaopt.

Results Figure 6a shows that Metaopt learns programs with lower costs given more training examples. After approximately 15 examples, Metaopt converges on the minimal cost program, refuting null hypothesis 1. Figure 6b shows similar results when measuring the runtimes of learned programs, and that the tree cost of a program corresponds to its time complexity.



(a) Tree costs (b) Program runtimes

Fig. 6: Figure (a) shows the median tree costs of programs learned by Metaopt when varying the number of training examples. Error bars represent the median absolute derivation. After approximately 15 examples, Metaopt converges on the minimal cost program. Figure (b) shows the corresponding runtimes.

5.2 Experiment 2: comparison with other systems

This experiment again revisits the *find duplicate* problem. The aim is to compare the tree costs and running times of programs learned by Metaopt to programs learned by Metagol and Metagol$_O$, where Metaopt minimises tree costs, Metagol minimises program size (Definition 4), and Metagol$_O$ minimises resource complexity (as described in Section 4.3). We test the null hypothesis:

Null hypothesis 2 Metaopt cannot learn programs with lower costs and lower running times than Metagol and Metagol$_O$.

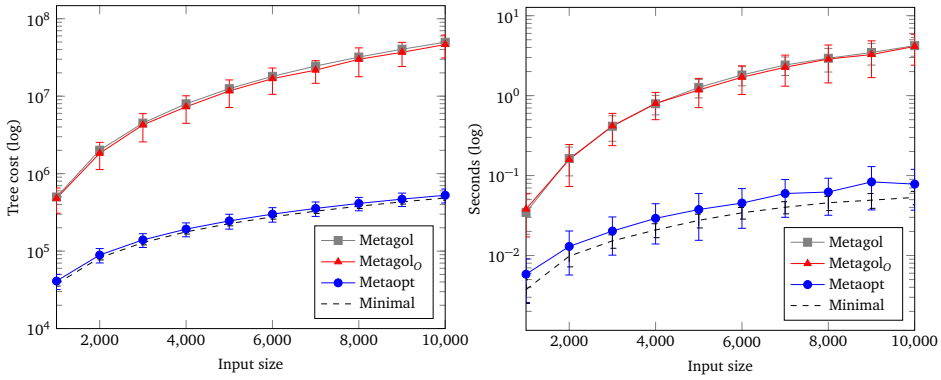
Materials We provide all three systems with the same background knowledge as in Experiment 1. We generate training examples in the same way as in Experiment 1. We generate testing examples using the same procedure but for fixed list sizes from the set $\{1000, 2000, \ldots, 10000\}$ to measure tree costs as the input grows.

Method Our experimental method is as follows:

1. Generate 20 training examples, half positive and half negative
2. Generate 100 testing examples, half positive and half negative
3. Learn a program p using the training examples with a timeout of 10 minutes.
4. Measure the tree cost and running time of p over the testing examples

We measure median tree costs and running times over 20 repetitions.

Results The log-lin plot in Figure 7a shows that Metaopt learns programs with lower tree costs than both Metagol and Metagol$_O$. The log-lin plot in Figure 7b shows similar results when measuring the runtimes of learned programs. Therefore, null hypothesis 2 is refuted both in terms tree costs and running times. Figures 2a and 2b show example programs learned by Metagol$_O$ and Metaopt respectively.



(a) Tree costs

(b) Program runtimes

Fig. 7: Figure (a) shows the median tree costs of learned *find duplicate* programs. Error bars represent the median absolute derivation. The costs of programs learned by Metaopt match those of the minimal cost program and are of order $O(n\ log\ n)$. By contrast, the programs learned by Metagol and Metagol$_O$ are of order $O(n^2)$. Figure (b) shows the corresponding runtimes.

5.3 Experiment 3: real-world string transformations

In [16] the authors evaluate Metagol on 17 real-world string transformation problems. Figure 8 shows problem *p01* where the goal is to learn a program that extracts the name from the input. This experiment explores whether Metaopt can learn minimal tree cost programs for these real-world problems.

Input	**Output**
My name is John.	John
My name is Bill.	Bill
My name is Josh.	Josh
My name is Albert.	Albert
My name is Richard.	Richard

Fig. 8: Examples for the *p01* string transformation problem.

Materials We provide Metaopt, Metagol, and Metagol$_O$ with the same background knowledge containing the *curry* and *chain* metarules (Figure 3) and the predicates: *is_letter/1*, *not_letter/1*, *is_uppercase/1*, *not_uppercase/1*, *is_number/1*, *not_number/1*, *is_space/1*, *not_space/1*, *tail/2*, *dropLast/2*, *reverse/2*, *filter/3*, *dropWhile/3*, and *takeWhile/3*.

Method The dataset from [16] contains five examples of each problem. We perform leave-two-out (keep-three-in) cross validation. We measure median program costs and running times over all trials. We set a timeout at 10 minutes.

Results Out of the 17 problems, Metagol, Metagol$_O$, and Metaopt learned different programs for 9 of them. Figure 9 shows the tree costs for the 9 problems, where Metaopt learns programs with lower costs in all cases, again refuting null hypothesis 2. For problem *p01*, the cost of the program learned by Metaopt (31) is half of that learned by Metagol (67) and Metagol$_O$ (86). Figure 10 shows example programs learned by the systems for problem *p01*. Although textually more complex, the program learned by Metaopt has a lower tree cost because it successively applies the *tail/2* predicate until it reaches the first letter of the name. By contrast, Metagol learns a program which uses the *dropWhile/3* predicate to recursively check whether the head symbol is uppercase, and if not drops the head element, which requires twice the amount of work. Because Metagol$_O$ only associates costs with dyadic predicates, it found a program which does not directly use any primitive dyadic predicates, and so has a resource cost of 0, yet is less efficient than the one found by Metaopt.

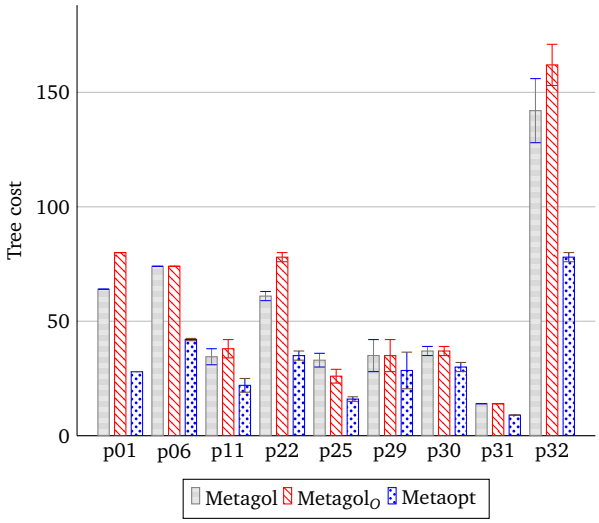


Fig. 9: Median tree costs of learned string transformation programs. Error bars represent the median absolute derivation. Metaopt learns lower tree cost programs than Metagol and Metagol$_O$ in all cases.

Metagol (a)

```
f(A,B):-tail(A,C),f1(C,B).
f1(A,B):-dropLast(A,C),f2(C,B).
f2(A,B):-dropWhile(A,B,not_uppercase).
```

Metagol$_O$ (b)

```
f(A,B):-f_1(A,C),f4(C,B).
f1(A,B):-f_2(A,C),f3(C,B).
f2(A,B):-filter(A,B,is_letter).
f3(A,B):-dropWhile(A,B,is_uppercase).
f4(A,B):-dropWhile(A,B,not_uppercase).
```

Metaopt (c)

```
f(A,B):-tail(A,C),f1(C,B).
f1(A,B):-f2(A,C),dropLast(C,B).
f2(A,B):-f3(A,C),f3(C,B).
f3(A,B):-tail(A,C),f4(C,B).
f4(A,B):-f5(A,C),f5(C,B).
f5(A,B):-tail(A,C),tail(C,B).
```

Fig. 10: Programs learned by Metagol, Metagol$_O$, and Metaopt for the *p01* string transformation problem.

5.4 Experiment 4: robot postman strategies

In [2], Metagol$_O$ is shown to learn robot strategies with lower resource complexities than Metagol. This experiment reproduces the postman experiment from [2] to show that Metaopt can simulate Metagol$_O$ by treating resource complexity as a specific case of the cost minimisation problem, i.e. by using the resource complexity implementation described in Section 4.2.2. We test the null hypotheses:

Null hypothesis 3 Metaopt cannot learn programs with lower resource complexities than Metagol.

Materials Imagine a humanoid robot postman learning to collect and deliver letters in a d sized one-dimensional space. In the initial state, the robot is at position 1 and n letters are to be collected. In the final state, the robot is at position 1 and n letters have been delivered to their intended destinations. The state is represented as a list of Prolog facts. The robot can perform primitive actions to transform the state: *move_right/2*, *move_left/2*, *pick_up_left/2*, *pick_up_right/2*, *drop_left/2*, *drop_right/2*, *take_letter/2*, *bag_letter/2*, and *give_letter/2*. All primitive actions have a cost of 1. The robot can also perform complex actions, which are defined in terms of primitive actions: *find_next_sender/2* and *find_next_recipient/2*, *go_to_start/2*, and *go_to_end/2*. The costs of complex actions are determined by their constituent primitive actions. The robot can take and carry a single

letter from a sender using the action *take_letter/2*. Alternatively, the robot can take a letter from a sender and place it a postbag using the action *bag_letter/2*, which allows the robot to carry multiple letters. We use the *chain* and *tailrec* metarules (Fig 3).

We generate a training example using the following procedure:

1. Select a random integer d from the interval $[10, 25]$ representing the number of houses.
2. Select a random integer n from the interval $[1, 5]$ representing the number of letters.
3. For each letter l, select random integers i and j from the interval $[1, d]$ representing the letter's start and end positions, such that $i \neq j$
4. Form an input state $s_1 =$

$$[pos(pman, 0), energy(0), pos(l_1, i_1), \ldots, pos(l_n, i_n), letter(l_1, i_1, j_1), \ldots, letter(l_n, i_n, j_n)]$$

5. Form an output state $s_2 =$

$$[pos(pman, \_), energy(\_), pos(l_1, j_1), \ldots, pos(l_n, j_n), letter(l_1, i_1, j_1), \ldots, letter(l_n, i_n, j_n)]$$

6. Form an example $f(s_1, s_2)$.

We generate a test example using the aforementioned procedure but with a fixed number of letters $d = 50$ and fixed number of letters n from the set $\{2, 4, \ldots, 20\}$ to measure the resource complexity as n grows.

Methods Our experimental method is as follows:

1. Generate 5 positive training and 100 positive testing examples
2. Learn a program p using the training examples with a timeout of 10 minutes.
3. Measure the resource complexity and running time of p over the testing examples

We measure median resource complexities of learned strategies over 20 trials.

Results Figure 11 shows that Metaopt learns strategies with lower resource complexities than Metagol, refuting null hypothesis 3. Figure 12 shows two strategies learned by Metagol (a) and Metaopt (b), able to handle any number of houses, any number of letters, and different start/end positions for the letters. Although the strategies are equal in their textual complexity, they differ in their resource complexity. The strategy learned by Metaopt (b) has lower resource complexity because it involves the use of the postbag to store letters, whereas strategy (a) does not.

6 Conclusions and further work

We have introduced Metaopt which extends MIL to support learning minimal cost logic programs. To find minimal cost programs, Metaopt uses iterative descent, which iteratively learns lower cost programs, each time further restricting the hypothesis space. We have shown (Theorem 1) that given sufficiently large numbers of examples, Metaopt converges on minimal cost programs, and that in practice (Experiment 1), only small numbers of examples are required. To learn minimal time-complexity programs, we introduced a cost function called tree cost (Definition 16), which is based on the size of a SLD-tree at the point of which a goal is proved by a logic program. Our experiments on the *find duplicate* problem show that Metaopt learns minimal cost programs given small numbers of examples. By contrast, Metagol and Metagol$_O$ both learn non-minimal cost

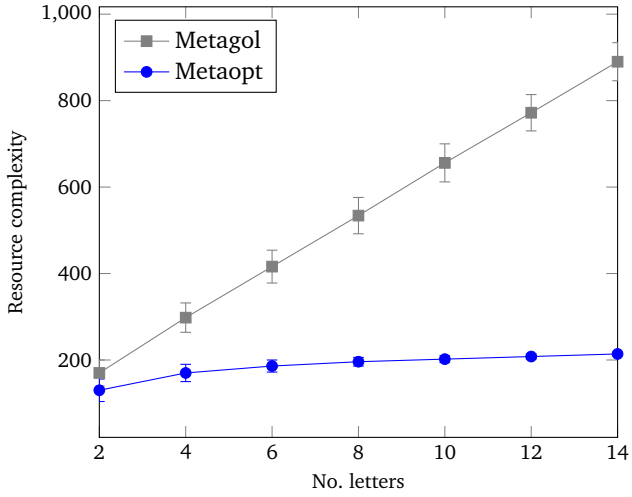


Fig. 11: Median resource complexity of learned postman strategies with varying numbers of letters for 50 places. Error bars represent the median absolute derivation.

Metagol (a)

```
f(A,B):-go_to_start(A,C),f1(C,B).
f(A,B):-f1(A,C),f(C,B).
f1(A,B):-find_next_recipient(A,C),f2(C,B).
f1(A,B):-find_next_sender(A,C),take_letter(C,B).
f2(A,B):-give_letter(A,C),go_to_start(C,B).
```

Metaopt (b)

```
f(A,B):-f1(A,C),f1(C,B).
f1(A,B):-f2(A,C),go_to_start(C,B).
f1(A,B):-f2(A,C),f1(C,B).
f2(A,B):-find_next_recipient(A,C),give_letter(C,B).
f2(A,B):-find_next_sender(A,C),bag_letter(C,B).
```

Fig. 12: Postman strategies learned by Metagol (a) and Metaopt (b) with resource complexities $O(n+d)$ and $O(nd)$ respectively.

programs with longer running times. Our experiments also show that Metaopt learns programs with lower costs than existing systems on some real-world string transformation problems. Finally, our experiments on learning robot strategies show that Metaopt can simulate Metagol$_O$ and learn minimal resource complexity robot strategies by treating resource complexity as a specific case of the cost minimisation problem.

6.1 Future work

Theorem 1 shows that Metaopt learns minimal cost programs given sufficient examples. The *find duplicate* experiment (Section 5.1) supported this result and showed that in practice only small numbers of examples (<20) are necessary. Future work should further test this result on other domains, such as learning from visual data or learn efficient taleo-reactive programs [27]. Likewise, in all of our experiments, we have assumed noise-free examples, which means that a learned program must be consistent with all examples. This assumption restricts MIL from being applied to noisy problems. To address this limitation, we could relax the requirement that a program must be consistent with all examples. One method, similar to the one used by [20], is to repeatedly learn programs from random subsets of the examples, and to then calculate confidence levels of the learned programs based on the size of the subsets and the number of repetitions.

Other complexity measures We have introduced techniques to learn minimal worst-case complexity programs. However, we are often interested in finding minimal average-case complexity programs, and future work should explore this topic. To do so, the framework in Section 3 would need to be adjusted to accommodate a different function to measure the cost of a program over a set of examples (Definition 3), which would effect the proof of convergence of Metaopt (Theorem 1). In this case, it would be desirable to analyse how many examples Metaopt would need to converge on the optimal average-case program. Another promising areas of future work include investigating whether Metaopt can learn minimal space-complexity programs or minimal power consumption programs.

Program complexity analysis Metaopt uses iterative descent to continually prune the hypothesis space of programs that are less efficient than already learned ones. However, this approach is inefficient when the first found program (in the first iteration of iterative descent) has a prohibitively high cost. For instance, suppose you are learning to sort lists and that the shortest program in the hypothesis space is permutation sort. Then in the first iteration of iterative descent, Metaopt would find permutation sort, which would require $O(n!)$ time. If the examples are large, then this approach would be impractical. To overcome this issue, iterative descent could start with a low program cost bound and then iteratively relax this bound until the first program is found. Once a program has been found, iterative descent could then work as it does now and search for more efficient programs by continually restricting the hypothesis space. Alternatively, we could estimate the tree complexity of a program by approximating the SLD-tree size [12].

Algorithm discovery We have used Metaopt to learn efficient programs, such as an efficient quicksort robot strategy and an efficient *find duplicate* program. However, although the learning techniques are novel, the learned programs are not, i.e. we have learned programs that we already knew about. In future work, we want to use Metaopt for *algorithm discovery*, where the goal is to learn programs that are both efficient and novel.

References

1. Lenore Blum and Manuel Blum. Toward a mathematical theory of inductive inference. *Information and Control*, 28(2):125–155, 1975.

2. Andrew Cropper and Stephen H. Muggleton. Learning efficient logical robot strategies involving composable objects. In *IJCAI*, pages 3423–3429. AAAI Press, 2015.
3. Andrew Cropper and Stephen H. Muggleton. Learning higher-order logic programs through abstraction and invention. In *IJCAI*, pages 1418–1424. IJCAI/AAAI Press, 2016.
4. Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.
5. Saumya K. Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. Lower bound cost estimation for logic programs. In *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, pages 291–305, 1997.
6. Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *Journal of Artificial Intelligence Research*, pages 25–71, 2003.
7. Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.
8. Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin G. Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):90–99, 2015.
9. Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, pages 253–302, 2001.
10. Elaine Kant. On the efficient synthesis of efficient programs. *Artif. Intell.*, 20(3):253–305, 1983.
11. S. Kaplan. Algorithmic complexity of logic programs. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 780–793, 1988.
12. Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *AAAI*, pages 1014–1019. AAAI Press, 2006.
13. J. E. Laird. Extending the soar cognitive architecture. *Frontiers in Artificial Intelligence and Applications*, pages 224–235, 2008.
14. Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Logics in Artificial Intelligence*, pages 311–325. Springer, 2014.
15. Leonid A. Levin. Randomness conservation inequalities; information and independence in mathematical theories. *Information and Control*, 61(1):15–37, 1984.
16. Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 525–530. IOS Press, 2014.
17. Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. In *IJCAI*, pages 542–551. William Kaufmann, 1979.
18. Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.
19. S. Moyle and S.H. Muggleton. Learning programs in the event calculus. In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh Inductive Logic Programming Workshop (ILP97)*, LNAI 1297, pages 205–212, Berlin, 1997. Springer-Verlag.
20. S.H. Muggleton, W-Z. Dai, C. Sammut, A. Tamaddoni-Nezhad, J. Wen, and Z-H. Zhou. Meta-interpretive learning from noisy images. *Machine Learning*, 2018. In Press.
21. Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286, 1995.
22. Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *ALT*, pages 368–381, 1990.
23. Stephen Muggleton, Ashwin Srinivasan, and Michael Bain. Compression, significance, and accuracy. In Derek H. Sleeman and Peter Edwards, editors, *Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK, July 1-3, 1992*, pages 338–347. Morgan Kaufmann, 1992.
24. Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014.
25. Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.
26. Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.
27. Nils J. Nilsson. Teleo-reactive programs for agent control. *J. Artif. Intell. Res. (JAIR)*, 1:139–158, 1994.
28. Ramón P. Otero. Induction of the indirect effects of actions by monotonic methods. In Stefan Kramer and Bernhard Pfahringer, editors, *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10-13, 2005, Proceedings*, volume 3625 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2005.

29. Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *J. Log. Program.*, 19/20:261–320, 1994.
30. G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969.
31. G.D. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1971.
32. Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
33. S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, New Jersey, 2010. Third Edition.
34. E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.
35. Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed*. MIT Press, 1994.
36. Phillip D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977.
37. Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.
38. Martijn van Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. In *Reinforcement Learning*, pages 3–42. Springer, 2012.
39. S. Vera. Induction of concepts in the predicate calculus. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*, pages 281–287, 1975.
40. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.
41. Zhao Xing, Yixin Chen, and Weixiong Zhang. Optimal strips planning by maximum satisfiability and accumulative learning. In *Proceedings of the International Conference on Autonomous Planning and Scheduling (ICAPS)*, pages 442–446, 2006.
42. Fangkai Yang, Piyush Khandelwal, Matteo Leonetti, and Peter Stone. Planning in answer set programming while learning action costs for mobile robots. *AAAI Spring 2014 Symposium on Knowledge Representation and Reasoning in Robotics (AAAI-SSS)*, 2014.
43. John M. Zelle and Raymond J. Mooney. Combining FOIL and EBG to speed-up logic programs. In *IJCAI*, pages 1106–1113. Morgan Kaufmann, 1993.