The ILASP System for Inductive Learning of Answer Set Programs

Mark Law[1,2], Alessandra Russo[2], and Krysia Broda[2]

[1] ILASP Limited, UK
mark@ilasp.com
[2] Department of Computing, Imperial College London, UK
{mark.law09, a.russo, k.broda}@imperial.ac.uk

Abstract. The goal of Inductive Logic Programming (ILP) is to learn a program that explains a set of examples in the context of some pre-existing background knowledge. Until recently, most research on ILP targeted learning Prolog programs. Our own ILASP system instead learns Answer Set Programs, including normal rules, choice rules and hard and weak constraints. Learning such expressive programs widens the applicability of ILP considerably; for example, enabling preference learning, learning common-sense knowledge, including defaults and exceptions, and learning non-deterministic theories. In this paper, we first give a general overview of ILASP's learning framework and its capabilities. This is followed by a comprehensive summary of the evolution of the ILASP system, presenting the strengths and weaknesses of each version, with a particular emphasis on scalability.

Keywords: Non-monotonic Inductive Logic Programming · Learning Answer Set Programs · Noise

1 Introduction

The ability to declaratively specify real-world problems and efficiently generate solutions from such specifications is of particular interest in both academia and industry [10, 9]. A typical paradigm is Answer Set Programming (ASP) [15, 3], which allows a problem to be described in terms of its specification, rather than requiring a user to define an algorithm to solve the problem. Its solvers are capable of constructing solutions from the specifications alone and, where needed, ranking solutions according to optimisation criteria. The interpretable nature of the ASP language also enables the generation of explanations, which is particularly relevant in AI-driven applications. Due to its rich language and efficient solving, ASP has been applied to a wide range of classical areas in AI – including planning, scheduling and diagnosis – and is increasingly being applied in industry [9]; for example, in decision support systems, automated product configuration and configuration of safety systems [10]. On the other hand, developing ASP specifications of real-world problems can be a difficult task for non-experts. Furthermore, the dynamic nature of the contexts in which real-world AI applications tend to operate can require the ASP specification of a problem to be regularly updated or revised. To widen the dissemination of ASP in practice, it is therefore crucial to develop methods that can automatically learn ASP specifications from examples of (partial) solutions to real-world problems. Such learning mechanisms could also provide ways for automatically evolving and revising ASP specifications in dynamic environments.

Within the last few years, we have addressed this problem and developed a novel system, called *Inductive Learning of Answer Set Programs* (ILASP) [21]. The theoretical framework underpinning the ILASP system differs from conventional approaches for Inductive Logic Programming (ILP), which are mainly focused on learning Prolog programs. Due to the declarative nature of ASP, the learning process in ILASP primarily targets learning the logical specification of a problem, rather than the procedure for solving that problem. Secondly, programs learned by ILASP can include extra types of rules that are not available in Prolog, such as choice rules and hard and weak constraints. Enabling the learning of these extra rules has opened up new applications, which were previously out of scope for ILP systems; for instance, learning weak constraints allows ILASP to learn a user's preferences from examples of which solutions the user prefers [22].

ILASP's learning framework has been proved to generalise existing frameworks and systems for learning ASP programs [24], such as the brave learning framework [30], adopted by almost all previous systems (e.g. XHAIL [28], ASPAL [6], ILED [16], RASPAL [2]), and the less common cautious learning framework [30]. Brave systems require the examples to be covered in at least one answer set of the learned program, whereas cautious systems find a program which covers the examples in every answer set. We showed in [24] that some ASP programs cannot be learned with either a brave or a cautious approach, and that to learn ASP programs in general, a combination of both brave and cautious reasoning is required. ILASP's learning framework enables this combination, and is capable of learning the full class of ASP programs [24]. ILASP's generality has allowed it to be applied to a wide range of applications, including event detection [25], preference learning [22], natural language understanding [4], learning game rules [7], grammar induction [18] and automata induction [11].

In this paper we give an introduction to ILASP's learning framework and its capabilities, demonstrating the various types of examples that ILASP can learn from, and describe the evolution of the ILASP system. This evolution has been driven by a need for efficiency with respect to various dimensions, including handling noisy data, large numbers of examples and large search spaces. We discuss the strengths and weaknesses of each variation of the ILASP system, explaining how each system improves on the efficiency of its predecessor. We conclude with a discussion of recent developments and future research directions.

2 Components of a Learning Task

ILASP is used to solve a *learning task*, which consists of three main components: the background knowledge, the mode bias and the examples. The *background* *knowledge* B is an ASP program, which describes a set of concepts that are already known before learning. ILASP accepts a subset[3] of ASP, consisting of normal rules, choice rules and hard and weak constraints. We use the term *rule* to refer to any of these four components.

The *mode bias* M (often called a language bias) is used to express the ASP programs that can be learned; for example, it specifies which predicates may be used in the head/body of learned rules, and how they may be used together. From M, it is possible to construct a (finite) set of rules S_M called the *rule space*[4], that contains every rule that is compatible with M. The power set of S_M, $\mathbb{P}(S_M)$, is called the *program space*, and contains the set of all ASP programs that can be learned.

The *examples* E describe a set of semantic properties that the learned program should satisfy. When the semantic property of an individual example $e \in E$ is satisfied, we say that e is *covered*. The goal of an ILP system, such as ILASP, is to find a program (often called a hypothesis) $H \in \mathbb{P}(S_M)$ such that $B \cup H$ (the combination of this program with the background knowledge) covers every example in E. Many ILP systems (including ILASP) follow the principle of Occam's Razor, that the simplest solution should be preferred, and therefore search for an *optimal* program, which is the shortest in terms of the number of literals.

Many ILP systems learn from (positive and negative) examples of atoms which should be true or false. This is because many ILP systems are targeted at learning Prolog programs, where the main "output" of a program is a query of a single atom. In ASP, the main "output" of a program is a set of answer sets. For this reason, ILASP learns from positive and negative examples of (partial) interpretations, which should or should not (respectively) be an answer set of the learned program. These examples are sufficient to learn any ASP program consisting of normal rules, choice rules and hard constraints (up to strong equivalence) [24]; however, it is not possible to learn weak constraints using only positive and negative examples, because they can only specify what should (or should not) be an answer set. Weak constraints do not have any effect on what is or is not an answer set – they only create a preference ordering over the answer sets. For this reason, ILASP allows a second type of example called an *ordering example*, the semantic property of which is a preference ordering over a pair of answer sets of $B \cup H$. Learning weak constraints corresponds to a form of *preference learning.*

2.1 Positive and Negative Examples

Consider a very simple setting, where we want to learn a program that describes the behaviour of three (specific) coins, by flipping them and observing which sides they land on. We can use a very simple mode bias to describe the rules we are allowed to learn. The predicates `heads/1` and `tails/1` are both allowed to appear

[3] For a formal definition of this subset, please see the ILASP manual (`http://www.ilasp.com/manual`).

[4] In other literature, the rule space is often called the *hypothesis space*.

in the head with a single argument, which is either a variable or constant of type `coin` (where there are three constants of type coin in the domain: `c1`, `c2` and `c3`). In the body, we can use three predicates (both positively and negatively): `heads/1`, `tails/1` and `coin/1`. The mode bias expressing this language is shown below.

```
#modeh(heads(var(coin))).         #modeh(heads(const(coin))).
#modeh(tails(var(coin))).         #modeh(tails(const(coin))).

#modeb(heads(var(coin))).         #constant(coin, c1).
#modeb(tails(var(coin))).         #constant(coin, c2).
#modeb(coin(var(coin))).          #constant(coin, c3).
```

We flip the coins twice, and see the following combinations of observations: {`heads(c1)`, `tails(c2)`, `heads(c3)`}, {`heads(c1)`, `heads(c2)`, `tails(c3)`}. We can encode these "observations" in ILASP using positive examples. Positive examples specify properties which should hold in at least one answer set of the learned program ($B \cup H$). The two observations are represented by the following two examples:

```
#pos({heads(c1), tails(c2), heads(c3)},
     {tails(c1), heads(c2), tails(c3)}).

#pos({heads(c1), heads(c2), tails(c3)},
     {tails(c1), tails(c2), heads(c3)}).
```

Each positive example contains two sets of ground atoms, called the inclusions and the exclusions (respectively). For a positive example to be covered, there must be at least one answer set of $B \cup H$ that contains all of the inclusions and none of the exclusions. In this case, these examples mean that there must be (at least) two answer sets, one which contains `heads(c1)`, `tails(c2)` and `heads(c3)`, and does not contain `tails(c1)`, `heads(c2)` and `tails(c3)`, and another answer set which contains `heads(c1)`, `heads(c2)` and `tails(c3)`, and does not contain `tails(c1)`, `tails(c2)` and `heads(c3)`. Although these particular examples completely describe the values of all coins, this does not need to be the case in general. Partial examples allow us to represent uncertainty; for example, we could have a fourth coin `c4` for which we do not know the value.

Together with the above examples, we can also give the following, very simple, background knowledge, which defines the set of coins we have:

```
coin(c1).    coin(c2).    coin(c3).
```

If we run this task in ILASP, then ILASP returns the following solution:

```
heads(V1) :- coin(V1), not tails(V1).
tails(V1) :- coin(V1), not heads(V1).
```

This program states that every coin must land on either `heads` or `tails`, but not both. Although the first coin `c1` has never landed on `tails` in the scenarios we have observed, ILASP has *generalised* to learn first-order rules that apply to all coins, rather than specific ground rules that only explain the specific instances we have seen. The ability to generalise in this way is a huge advantage of ILP systems over other forms of machine learning, because it usually means that ILP techniques require very few examples to learn general concepts. Note that both positive examples are required for ILASP to learn this general program. Neither positive example on its own is sufficient because in both cases there is a shorter program that explains the example – the set of facts {`heads(c1)`. `tails(c2)`. `heads(c3)`.} covers the first example and similarly the set of facts {`heads(c1)`. `heads(c2)`. `tails(c3)`.} covers the second.

It may be that after many more observations, we still have not witnessed `c1` landing on `tails`, and we could be convinced that it never will. In this case, we can use ILASP's negative examples to specify that there should be no answer set that contains `tails(c1)`. This example is expressed in ILASP as follows:

```
#neg({tails(c1)}, {}).
```

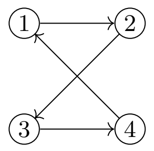
Given this extra example, ILASP learns the slightly larger program:

```
heads(V1) :- coin(V1), not tails(V1).
tails(V1) :- coin(V1), not heads(V1).
heads(c1).
```

This program states that all coins must land on either `heads` or `tails`, but not both, except for `c1`, which can only land on `heads`. Note that negative examples often cause ILASP to learn programs with rules that eliminate answer sets. In this case, the fact `heads(c1)` eliminates all answer sets that contain `tails(c1)`. Negative examples are often used to learn constraints. The constraint "`:-tails(c1)`." would have has the same effect; however, it is not permitted because the mode bias does not allow constants to be used in the body of a rule.

Context-dependent Examples. Positive and negative examples of partial answer sets are targeted at learning a fixed program, $B \cup H$. When ASP is used in practice, however, a program representing a general problem definition is often combined with another set of rules (usually just facts) describing a particular instance of the problem to be solved. For instance, a general program defining what it means for a graph to be Hamiltonian (i.e. the general problem definition) can be combined with a set of facts describing a particular graph (i.e. a problem instance). The combined program is satisfiable if and only if the graph represented by the set of facts is Hamiltonian. The *context-dependent* behaviour of the general Hamilton program cannot be captured by positive and negative examples of partial answer sets. Instead, we need an extension, called a *context-dependent example*. This allows each example to come with its own extra bit of background knowledge, called a context C, which applies only to that example.

```
#pos({}, {}, {
  node(1..4).
  edge(1, 2).
  edge(2, 3).
  edge(3, 4).
  edge(4, 1).
}).
```



```
#neg({}, {}, {
  node(1..4).
  edge(1, 2).
  edge(2, 1).
  edge(2, 3).
  edge(3, 4).
  edge(4, 2).
}).
```

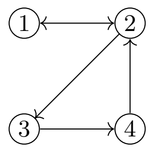


Fig. 1. One positive and one negative example of Hamiltonian graphs. On the left is the ILASP representation of the example, and on the right is the corresponding graph.

It is now $B \cup H \cup C$ that has to satisfy the semantic properties of the example, rather than $B \cup H$. In ILASP, the context of an example is expressed by adding an extra set to the example, containing the context.

Consider the two context-dependent examples in Figure 1. Both examples have empty inclusions and exclusions. In the case of a positive example, this simply means that there must exist at least one answer set of $B \cup H \cup C$ – any answer set is consistent with the empty partial interpretation – and in the case of a negative example, it means that there should be no answer set of $B \cup H \cup C$. Given a sufficient number of examples of this form, ILASP can be used to learn a program that corresponds to the definition of a Hamiltonian graph; i.e. the program $B \cup H \cup C$ is satisfiable if and only if the set of facts C represents a Hamiltonian graph. The full program learned by ILASP is:

```
0 { in(V0, V1) } 1 :- edge(V0, V1).
reach(V0) :- in(1, V0).
reach(V1) :- reach(V0), in(V0, V1).
:- not reach(V0), node(V0).
:- V1 != V2, in(V0, V2), in(V0, V1).
```

This example shows the high expressive power of ILASP, compared to many other ILP systems, which are only able to learn definite logic programs. In this case, ILASP has learned a choice rule, constraints and a recursive definition of reachability. The full learning task, `hamilton.las`, used to learn this program is available online.[5]

[5] For instructions on how to install ILASP, see `http://www.ilasp.com`. All learning tasks discussed in this section are available at `http://www.ilasp.com/research`.

2.2 Ordering Examples

Positive and negative examples can be used to learn any ASP program consisting of normal rules, choice rules and hard constraints.[6] As positive and negative examples can only express what should or should not be an answer set of the learned program, they cannot be used to learn weak constraints, which do not affect what is or is not an answer set. Weak constraints create a preference ordering over the answer sets of a program, so in order to learn them we need to give examples of this preference ordering – i.e. examples of which answer sets should be preferred to which other answer sets. These *ordering examples* come in two forms: *brave orderings*, which express that at least one pair of answer sets that satisfy the semantic properties of a pair of positive examples are ordered in a particular way; and *cautious orderings*, which express that every such pair of answer sets should be ordered in that way.

Consider a scenario in which a user is planning journeys from one location to another. All journeys consist of several legs, in which the user may take various modes of transport. Other known attributes of the journey legs are the distance of the leg, and the crime rating of the area (which ranges from 0 – no crime – to 5 – extremely high). By offering the user various journey options, and observing their choices, we can use ILASP to learn the preferences the user is using to make such choices. The options a user could take can be represented using context-dependent examples. Four such examples are shown below. Note that the first argument of the example is a unique identifier for the example. This identifier is optional, but is needed when expressing ordering examples.

```
#pos(eg_a, {}, {}, {
  leg_mode(1, walk).
  leg_crime_rating(1, 2).
  leg_distance(1, 500).
  leg_mode(2, bus).
  leg_crime_rating(2, 4).
  leg_distance(2, 3000).
}).

#pos(eg_b, {}, {}, {
  leg_mode(1, bus).
  leg_crime_rating(1, 2).
  leg_distance(1, 4000).
  leg_mode(2, walk).
  leg_crime_rating(2, 5).
  leg_distance(2, 1000).
}).
```

```
#pos(eg_c, {}, {}, {
  leg_mode(1, bus).
  leg_crime_rating(1, 2).
  leg_distance(1, 400).
  leg_mode(2, bus).
  leg_crime_rating(2, 4).
  leg_distance(2, 3000).
}).

#pos(eg_d, {}, {}, {
  leg_mode(1, bus).
  leg_crime_rating(1, 5).
  leg_distance(1, 2000).
  leg_mode(2, bus).
  leg_crime_rating(2, 1).
  leg_distance(2, 2000).
}).
```

[6] This result holds, up to strong equivalence, which means that given any such ASP program P, it is possible to learn a program that is strongly equivalent to P [24].

By observing a user's choices, we might see that the user prefers the journey represented by `eg_a` to the one represented by `eg_b`. This can be expressed in ILASP using an ordering example:

```
#brave_ordering(eg_a, eg_b, <).
```

This states that at least one answer set of $B \cup H \cup C_a$ must be preferred to at least one answer set $B \cup H \cup C_b$ (where C_a and C_b are the contexts of the examples `eg_a` and `eg_b`, respectively, and B, in this simple case, is empty). The final argument of the brave ordering is an operator, which says how the answer sets should be ordered. The operator $<$ means "strictly preferred". It is also possible to use any of the other binary comparison operators: $>$, $<=$, $>=$, $=$ or $!=$. For instance, the following example states that the journeys represented by `eg_c` and `eg_d` should be equally preferred.

```
#brave_ordering(eg_c, eg_d, =).
```

By using several such ordering examples, it is possible to learn weak constraints corresponding to a user's journey preferences. For example, the learning task `journey.las` (available online) causes ILASP to learn the following set of weak constraints:

```
:~ leg_mode(L, walk), leg_crime_rating(L, C), C > 3.[1@3, L, C]
:~ leg_mode(L, bus).[1@2, L]
:~ leg_mode(L, walk), leg_distance(L, D).[D@1, L, D]
```

These weak constraints represent that the user's top priority is to minimise the number of legs of the journey in which the user must walk through an area with a high crime rating; their next priority is to minimise the number of buses the user must take; and finally, their lowest priority is to minimise the total walking distance of their journey.

Note that in the given scenario there is always a single answer set of $B \cup H \cup C$ for each of the contexts C, meaning that brave and cautious orderings coincide. When $B \cup H \cup C$ may have multiple answer sets, the distinction is important, and cautious orderings are much stronger than brave orderings, expressing that the preference ordering holds universally over all pairs of answer sets that meet the semantic properties of the positive examples.

2.3 Noisy Examples

Everything presented so far in this paper assumes that all examples are correctly labelled, and therefore that all examples should be covered by the learned program. In real applications, of course, this is often not the case; examples may be noisy (i.e. mislabelled), and so finding a program that covers all examples may not be possible, or even desirable (as this might be overfitting on the examples). In ILASP, each example can be given a *penalty*, which is a cost for not covering that example. The search for an optimal learned program now searches for a program that minimises $|H| + cost$, where $|H|$ is the length of the program and *cost* is the sum of the penalties of all examples that are not covered by the learned program. We have used this approach to noise to apply ILASP to a wide range of real world problems, including event detection [25], sentence chunking [25], natural language understanding [4] and user preference learning [25].

The function $|H| + cost$ is one example of a *scoring function.* Most systems, including ILASP, come with a built-in scoring function that cannot be modified, but in recent work [19], we have developed a new system that allows the user to define their own scoring function, allowing a custom (domain-specific) interpretation of optimality.

3 Evolution of the ILASP system

Although we refer to ILASP as a single system, in reality it is a collection of algorithms, with each algorithm developed to address a scalability weakness of its predecessor.[7] Table 1 considers various "dimensions" of learning tasks and shows which ILASP algorithms scale with respect to each of these dimension.

	Scales with			
Algorithm	Any Negative Examples?	# of Examples	Level of Noise	Size of S_M
ILASP1	No	No	No	No
ILASP2	Yes	No	No	No
ILASP2i	Yes	Yes	No	No
ILASP3	Yes	Yes	Yes	No

Table 1. A summary of the scalability of each ILASP system, with respect to various dimensions of learning tasks.

Note that although newer versions of ILASP scale with respect to various dimensions of learning tasks, none of the current ILASP systems scales with respect to large rule spaces; however, this is being addressed in current work (for more details, see the next section).

3.1 ILASP1 and ILASP2

As depicted in Figure 2, the first two ILASP systems both have three main phases: (1) pre-processing; (2) solving; and (3) post-processing. In the first phase, ILASP1 and ILASP2 map the input learning task into an ASP program. Next, this ASP program is solved by the Clingo ASP solver [14, 12]. Finally, the answer set returned by Clingo is post-processed to extract the learned program.

[7] The learning framework has also been expanded with each new algorithm; however, older algorithms have been updated so that every ILASP algorithm supports the most general version of the learning framework.

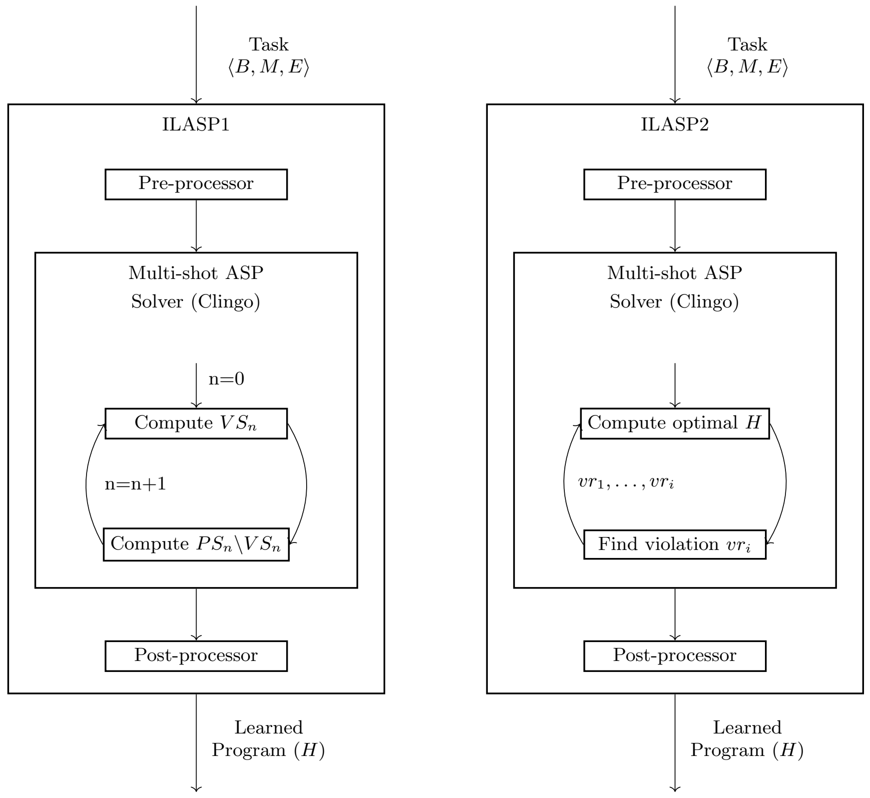


Fig. 2. ILASP1 and ILASP2. PS_n and VS_n denote, respectively, the positive and violating hypotheses of length n and $vr_1, \ldots, vr_i$ are the current violating reasons.

The procedures of the ILASP1 [20] and ILASP2 [22] algorithms are encoded using Clingo's built-in scripting feature, which allows a technique called *multi-shot* solving [13] to be used. Multi-shot solving enables a program to be solved iteratively, each time adding new parts to the program (or removing existing ones). The difference between the first two ILASP algorithms is in the multi-shot procedure. Both systems rely on the concepts of *positive hypotheses* – programs that cover all of the positive examples – and *violating hypotheses* – which also cover all of the positive examples, but do not cover at least one negative example. Starting at length $n = 0$, in each iteration, ILASP1 computes all the violating hypotheses of length n, converts each violating hypothesis to a constraint, which is added to the program, and then searches for a positive hypothesis of length n that does not violate any of the computed constraints (i.e. a program of length n which covers all of the examples). If there is such a program, it is returned; otherwise, n is incremented and the next iteration begins. As there can be a large number of violating hypotheses, and because there is one constraint per violating hypothesis, this process can be very inefficient if there is at least one negative example. ILASP2, on the other hand, computes a single (optimal) positive solution H in each iteration. If H is a violating hypothesis, then it extracts a "violating reason" vr, which explains why H is violating. It then encodes vr into a set of ASP rules, which are added to the program in order to rule out not only H, but also any other program which is violating for the same reason vr. Compared with ILASP1, ILASP2 adds far less to the program in each iteration, and often requires fewer iterations (as the same violating reason will often apply to many violating hypotheses), leading to orders of magnitude of improvement in performance on tasks with negative examples.

3.2 ILASP2i

The number of rules in the grounding of the ASP encoding used by ILASP1 and ILASP2 is proportional to the number of examples in the learning task. As the size of the grounding of an ASP program is one of the major factors in how long it takes for Clingo to solve that program, this means that ILASP1 and ILASP2 do not scale with respect to the number of examples.

In real datasets, there is often considerable overlap between the concepts required to cover several different examples. In other words, there are *classes* of examples such that each example in a class is covered by exactly the same programs as every other example in the class. In a non-noisy setting (where all examples must be covered), only one example per class is actually required, and all other examples are "irrelevant". The idea behind ILASP2i is to construct a subset of the examples called the *relevant examples*, which is often significantly smaller than the full set of examples, but nonetheless still forces ILASP to learn the correct program. The construction of the relevant examples is achieved by interleaving the search for an optimal program that covers the (partially constructed) set of relevant examples with a second search for a new relevant example – an example that is not covered by the current program H. This interleaving is illustrated in Figure 3. At the start of the process, the program H is set to be empty, because at this point this is the shortest program that covers the (empty) set of relevant examples. In each iteration, ILASP2i searches for a new relevant example, and if it finds one, it searches for a new program (updating H) that covers all relevant examples found so far, using ILASP2 to perform the search. If no relevant example exists, then H is an optimal solution of the task, and is returned. An example of the ILASP2i procedure, based on the coin learning task from the previous section, is shown in Figure 4.

The experiments in [23] demonstrate that ILASP2i can be over two orders of magnitude faster than ILASP2 on tasks with hundreds of examples. The reason is that the call to ILASP2 calls Clingo with an ASP program whose grounding is only proportional to the number of relevant examples, rather than to the full set of examples. Note that although the call to Clingo in the relevant example search considers all examples, its grounding is not proportional to the number of examples and it only requires single-shot solving in Clingo, meaning that each call to Clingo is relatively cheap compared to the Clingo execution in ILASP2.

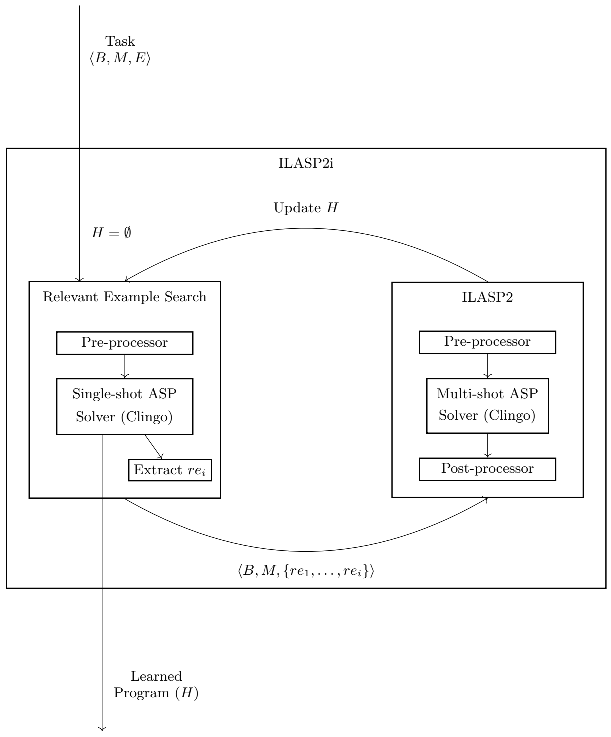


Fig. 3. ILASP2i. The relevant examples computed so far are denoted $re_1, \ldots, re_i$.

Relation to other approaches. ILASP1 and ILASP2 are examples of *batch learners*, which consider all examples simultaneously. Some older ILP systems, such as ALEPH [31], Progol [26] and HAIL [29], incrementally consider each positive example in turn, employing a *cover loop*. The idea behind a cover loop is that the algorithm starts with an empty program H and, in each iteration, adds new rules to H such that a single positive example e is covered, and none of the negative examples are covered. This approach does not work in a non-monotonic setting, as new rules could "undo" the coverage of previously covered examples. For this reason, most ASP learners are batch learners (e.g. [28, 6]). ILASP2i's method of using relevant examples can essentially be thought of as a non-monotonic version of the cover loop. There are three main differences:

1. In cover loop approaches, in each iteration a previous program H is extended with extra rules, giving a new program H' that contains H. In ILASP2i, a completely new program is learned in each iteration. This not only resolves the issue of non-monotonicity, but is also necessary to guarantee that optimal programs are computed. Many cover loop approaches make no guarantee about the optimality of the final learned program.

```
% Background Knowledge

coin(c1).
coin(c2).
coin(c3).

% Examples

#pos(eg1, {heads(c1), tails(c2), heads(c3)},
          {tails(c1), heads(c2), tails(c3)}).

#pos(eg2, {heads(c1), heads(c2), tails(c3)},
          {tails(c1), tails(c2), heads(c3)}).

#pos(eg3, {tails(c1), heads(c2), tails(c3)},
          {heads(c1), tails(c2), heads(c3)}).

#pos(eg4, {tails(c1), tails(c2), tails(c3)},
          {heads(c1), heads(c2), heads(c3)}).


% Mode bias

#modeh(heads(var(coin))).
#modeh(tails(var(coin))).
#modeh(heads(const(coin))).
#modeh(tails(const(coin))).

#modeb(heads(var(coin))).
#modeb(tails(var(coin))).
#modeb(coin(var(coin))).

#constant(coin, c1).
#constant(coin, c2).
#constant(coin, c3).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Iteration 1
%%
%% Hypothesis:
%%
%% Searching for an uncovered example...
%% The hypothesis does not cover the example: eg1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Iteration 2
%%
%% Searching for a hypothesis that covers the
%% examples in { eg1 }.
%%
%% Hypothesis:
%%
%% tails(c2).  heads(c1).  heads(c3).
%%
%% Searching for an uncovered example...
%% The hypothesis does not cover the example: eg2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Iteration 3
%%
%% Searching for a hypothesis that covers the
%% examples in { eg1, eg2 }.
%%
%% Hypothesis:
%%
%% heads(V1) :- coin(V1), not tails(V1).
%% tails(V1) :- coin(V1), not heads(V1).
%%
%% Searching for an uncovered example...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Solution found:
heads(V1) :- coin(V1), not tails(V1).
tails(V1) :- coin(V1), not heads(V1).
```

Fig. 4. On the left, an extension of the coin learning task from Section 2 (with more examples), and on the right, the output from ILASP2i. In the first iteration, ILASP2i searches for an example that is not covered by the empty program, and finds `eg1`. In the second iteration, it finds a very specific program that covers `eg1`, and then finds the second relevant example `eg2`. Next, it searches for a program that covers both relevant examples, and finds a more general program. As this program covers all examples, no further relevant examples are computed, and the process terminates.

2. In ILASP2i, the set of relevant examples is maintained and used in every iteration, whereas in cover loop approaches, only one example is considered per iteration.
3. In cover loop approaches, once an example has been processed, even if it did not cause any changes to the current program H, it is guaranteed to be covered by any future program H' and so it is not checked again. In ILASP2i, this is not the case. ILASP2i performs the search for relevant examples on the full set of examples, even if some were previously known to be covered.

ILASP2i is also somewhat similar to *active learning* algorithms, such as L^* [1]. Active learners are able to query an *oracle* as to whether what they have learned is correct. The oracle is then able to provide counterexamples to aid further learning. ILASP2i's set of relevant examples are very similar to the counterexamples provided by the oracle. The main difference between the two

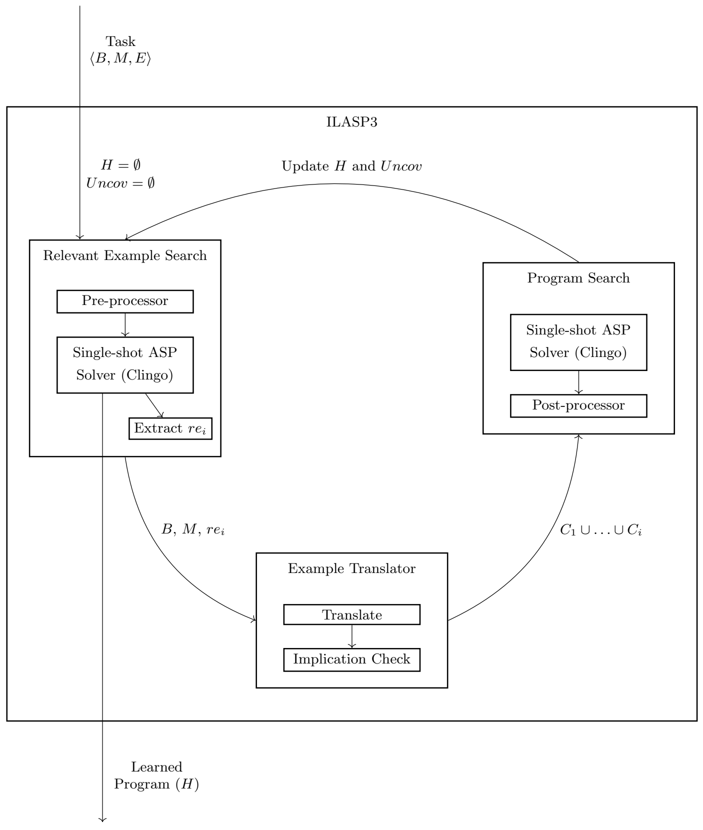


Fig. 5. The ILASP3 algorithm.

approaches is that in active learning, the oracle is assumed to know the correct definition of the concept being learned, whereas in ILASP2i, this is not the case, and the search for relevant examples is only over the provided training examples.

3.3 ILASP3

Although the concept of relevant examples allows ILASP2i to scale far better than ILASP2 with respect to the number of examples, this only holds in a non-noisy setting, where all examples must be covered. When examples can be noisy, finding a single relevant example only means that a penalty must be paid if the example is not covered. Many relevant examples (of the same class) may need to be found before their total penalty makes it “worth” covering the examples. For this reason, the final set of relevant examples is usually much larger in noisy settings, which significantly reduces the technique’s impact on scalability; in fact, ILASP2i is often even slower than ILASP2 on tasks with large numbers of potentially noisy examples (i.e. large numbers of examples with finite penalties) [17].

ILASP3 uses a novel method of *translating* an example into a set of *coverage constraints* over the solution space. The intuition of these coverage constraints is that they give a list of conditions which are satisfied by exactly those programs that cover the examples (e.g. the learned program must include at least one of a certain set of rules, or none of another set of rules). The benefit of having these constraints is twofold. Firstly, finding the optimal program that conforms to the constraints can be performed using a single-shot ASP call rather than a multi-shot call, as in ILASP2i, meaning that in ILASP3 the search for the optimal program is computationally much cheaper. This is because most of the work is done in the translation procedure (which does, itself, use a multi-shot call to Clingo). Secondly, once the constraints for one example have been computed, it is possible to check whether these constraints are necessary for other examples to be covered. This second benefit is the main reason why ILASP3 performs so much better than ILASP2i on tasks with noisy examples. After a relevant example is found (and translated), it is known that the computed constraints must be satisfied by the learned program, otherwise a whole set of examples will not be covered (and the penalties of each example in this set must be paid), rather than just the single relevant example as in ILASP2i. This can significantly reduce the number of iterations required by ILASP3, compared with ILASP2i.

Figure 5 depicts the procedure of ILASP3. It is similar in structure to ILASP2i, and similarly interleaves the program search with a search for relevant examples. The main difference is the addition of the *Example Translator*, which has two steps: firstly, it translates the relevant examples into a set of constraints on the solution space; and secondly, in the *implication check* step, it checks which other examples would be guaranteed to not be covered if the coverage constraints were not satisfied – i.e. for which other examples the coverage constraints are necessary conditions. The coverage constraints give an approximation of the coverage and score of every program in the program space. This approximation of a program's score is always guaranteed to be less than or equal to the program's real score, as it will only overestimate the program's coverage (if the program violates a coverage constraint, then it is known not to cover the corresponding examples). The program search computes the optimal program H with respect to the approximation of the score. When the approximation of the score of H is correct (i.e. the approximation of the coverage of H is equal to the true coverage of H), H is guaranteed to be an optimal solution of the learning task, and is returned. Checking whether the approximation is correct is performed by the relevant example search. As the approximation never underestimates the coverage of H, it suffices to only search for a relevant example within the set of examples that the approximation says H should cover. To facilitate this, in addition to returning H, the program search also returns the set of examples, $Uncov$, which are known not to be covered by H (according to the coverage constraints). The relevant example search is then within $E\backslash Uncov$, rather than the full example set E.

In addition to the procedure described in this section, ILASP3 has several other optional features, designed to boost performance on certain types of task. For more information, please see [17].

4 Current and Future Work

4.1 Conflict-driven ILP and ILASP4

Meta-level ILP systems, such as TAL [5], ASPAL [6] and Metagol [27, 8], encode an ILP task as a fixed meta-level logic program, which is solved by an off-the-shelf Prolog or ASP solver, after which the meta-level solution is translated back to an (object-level) inductive solution of the ILP task.

At first glance, the earliest ILASP systems (ILASP1 and ILASP2) may seem to be meta-level systems, and they do indeed involve encoding a learning task as a meta-level ASP program; however, they are actually in a more complicated category. Unlike "pure" meta-level systems, the ASP solver is not invoked on a fixed program, and is instead (through the use of multi-shot solving) incrementally invoked on a program that is growing throughout the execution.

With each new version, ILASP has shifted further away from pure meta-level approaches, towards a new category of ILP system, which we call *conflict-driven*. Conflict-driven ILP systems, inspired by conflict-driven SAT and ASP solvers, iteratively construct a set of constraints on the solution space – where the term constraint is used very loosely to mean anything that partitions the solution space into one partition that satisfies the constraint and another that does not – which must be satisfied by any inductive solution. In each iteration, the solver finds a program H that satisfies the current constraints, then searches for a *conflict* C, which corresponds to a reason why H is not an (optimal) inductive solution. If none exists, then H is returned; otherwise, C is converted to a new constraint which the next programs must satisfy.

In some sense ILASP2i is already a conflict-driven ILP system, where the relevant examples in each iteration are the conflicts, although it is not really in the spirit of a true conflict-driven system as the constraint generated in each iteration is that one of the examples must be covered, which was already obvious from the original task. ILASP3 is arguably the first truly conflict-driven ILP system, as it translates the relevant example (the conflict) into a set of constraints on the solution space; however, unlike conflict-driven SAT and ASP approaches the constraints can be extremely large and expensive to compute, especially when the program space is large. The issue stems from the fact that the constraints are both sufficient and necessary for the example to be covered (i.e. the example is covered if and only if the constraints are satisfied). ILASP4, which is currently in development, relaxes this and computes constraints which are only guaranteed to be necessary (but may not be sufficient) for the example to be covered. This may mean that the same relevant example is found twice, leading to more iterations, but each iteration will be considerably less expensive, and the constraints constructed in each iteration will be significantly smaller.

4.2 FastLAS

Although each ILASP system has improved scalability with respect to several dimensions, one bottleneck that remains is the size of the rule space. This is because every version of ILASP begins by computing the rule space in full. FastLAS [19] is a new algorithm that solves a restricted version of ILASP's learning task (currently with no recursion, only observational predicate learning and no predicate invention). Rather than generating the rule space in full, FastLAS computes a much smaller subset of the rule space that is guaranteed to contain at least one optimal solution of the task (called an *OPT-sufficient subset*). As this OPT-sufficient subset is often many orders of magnitude smaller than the full rule space, FastLAS is far more scalable than ILASP. Due to FastLAS's restrictions, once it has computed the OPT-sufficient subset, it is able to solve the task in one (single-shot) call to Clingo. FastLAS2, which is currently in development, will lift the restrictions and replace the call to Clingo with a call to ILASP, thus enabling ILASP to take advantage of FastLAS's increased scalability.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987)
2. Athakravi, D., Corapi, D., Broda, K., Russo, A.: Learning through hypothesis refinement using answer set programming. In: International Conference on Inductive Logic Programming. pp. 31–46. Springer (2013)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12), 92–103 (2011)
4. Chabierski, P., Russo, A., Law, M., Broda, K.: Machine comprehension of text using combinatory categorial grammar and answer set programs. CEUR Workshop Proceedings (2017)
5. Corapi, D., Russo, A., Lupu, E.: Inductive logic programming as abductive search. In: ICLP (Technical Communications). pp. 54–63 (2010)
6. Corapi, D., Russo, A., Lupu, E.: Inductive logic programming in answer set programming. In: Inductive Logic Programming, pp. 91–97. Springer (2012)
7. Cropper, A., Evans, R., Law, M.: Inductive general game playing. Machine Learning pp. 1–42 (2019)
8. Cropper, A., Muggleton, S.H.: Metagol system (2016), `https://github.com/metagol/metagol`
9. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Magazine **37**(3), 53–68 (2016)
10. Falkner, A., Friedrich, G., Schekotihin, K., Taupe, R., Teppan, E.C.: Industrial applications of answer set programming. KI-Künstliche Intelligenz **32**(2-3), 165–176 (2018)
11. Furelos-Blanco, D., Law, M., Russo, A., Broda, K., Jonsson, A.: Induction of subgoal automata for reinforcement learning. In: AAAI. Association for the Advancement of Artificial Intelligence (2020)
12. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)

13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019)
14. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications **24**(2), 107–124 (2011)
15. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. vol. 88, pp. 1070–1080 (1988)
16. Katzouris, N., Artikis, A., Paliouras, G.: Incremental learning of event definitions with inductive logic programming. Machine Learning **100**(2-3), 555–585 (2015)
17. Law, M.: Inductive Learning of Answer Set Programs. Ph.D. thesis, Imperial College London (2018)
18. Law, M., Russo, A., Bertino, E., Broda, K., Lobo, J.: Representing and learning grammars in answer set programming. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 2919–2928 (2019)
19. Law, M., Russo, A., Bertino, E., Broda, K., Lobo, J.: FastLAS: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In: AAAI. Association for the Advancement of Artificial Intelligence (2020)
20. Law, M., Russo, A., Broda, K.: Inductive learning of answer set programs. In: Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings. pp. 311–325 (2014)
21. Law, M., Russo, A., Broda, K.: The ILASP system for learning answer set programs. `http://www.ilasp.com/` (2015)
22. Law, M., Russo, A., Broda, K.: Learning weak constraints in answer set programming. Theory and Practice of Logic Programming **15**(4-5), 511–525 (2015)
23. Law, M., Russo, A., Broda, K.: Iterative learning of answer set programs from context dependent examples. Theory and Practice of Logic Programming **16**(5-6), 834–848 (2016)
24. Law, M., Russo, A., Broda, K.: The complexity and generality of learning answer set programs. Artificial Intelligence **259**, 110–146 (2018), `http://www.sciencedirect.com/science/article/pii/S000437021830105X`
25. Law, M., Russo, A., Broda, K.: Inductive learning of answer set programs from noisy examples. Advances in Cognitive Systems (2018)
26. Muggleton, S.: Inverse entailment and Progol. New Generation Computing **13**(3-4), 245–286 (1995)
27. Muggleton, S., Lin, D.: Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. pp. 1551–1557. AAAI Press (2013)
28. Ray, O.: Nonmonotonic abductive inductive learning. Journal of Applied Logic **7**(3), 329–340 (2009)
29. Ray, O., Broda, K., Russo, A.: Hybrid abductive inductive learning: A generalisation of Progol. In: Inductive Logic Programming, pp. 311–328. Springer (2003)
30. Sakama, C., Inoue, K.: Brave induction: a logical framework for learning from incomplete information. Machine Learning **76**(1), 3–35 (2009)
31. Srinivasan, A.: The Aleph manual. Machine Learning at the Computing Laboratory, Oxford University (2001)