Logic-based Learning of Answer Set Programs

Mark Law, Alessandra Russo, and Krysia Broda

Department of Computing, Imperial College London, UK

Abstract. Learning interpretable models from data is stated as one of the main challenges of AI. The goal of logic-based learning is to compute interpretable (logic) programs that explain labelled examples in the context of given background knowledge. This tutorial introduces recent advances of logic-based learning, specifically learning non-monotonic logic programs under the answer set semantics. We introduce several learning frameworks and algorithms, which allow for learning highly expressive programs, containing rules representing non-determinism, choice, exceptions, constraints and preferences. Throughout the tutorial, we put a strong emphasis on the expressive power of the learning systems and frameworks, explaining why some systems are incapable of learning particular classes of programs.

Keywords: Non-monotonic Inductive Logic Programming · Generality of learning frameworks · Learning Answer Set Programs.

1 Introduction

Over the last decade we have witnessed a growing interest in Machine Learning. In recent years Deep Learning has been demonstrated to achieve high-levels of accuracy in data analytics, signal and information processing tasks, bringing transformative impact in domains such as facial, image, speech recognition, and natural language processing. They have best performance on computational tasks that involve large quantities of data and for which the labelling process and feature extraction would be difficult to handle. However, they also suffer from two main drawbacks, which are crucial in the context of cognitive computing. They are not capable of supporting AI solutions that are good at more than one task. They are very effective when applied to single specific tasks (e.g. recognition of specific clues, objects in images, natural language translation). But applying the same technology from one task to another within the same class of problems would often require retraining, causing the system to possibly *forget* how to solve a previously learned task. Secondly, and most importantly, they are not *transparent*. Operating primarily as black boxes, deep learning approaches are not amenable to human inspection and human feedbacks, and the learned models are not explainable, leaving the humans agnostic of the cognitive and learning process performed by the system. This lack of transparency hinders human comprehension, auditing of the learned outcomes, and human active engagement into the learning and reasoning processes performed by the AI

systems. This has become an increasingly important issue in view of the recent General Data Protection Regulation (GDPR) which requires actions taken as a result of a prediction from a learned model to be justified.

Within the last decade, there has been a growing interest in Machine Learning approaches whose learned models are explainable and human interpretable. The last ten years have witnessed a tremendous advancement in the field of logic-based machine learning, also referred to as Inductive Logic Programming (ILP) [28, 30], where the goal is the automated acquisition of knowledge (expressed as a logic program) from given (labelled) examples and existing background knowledge. The main advantage of these machine learning approaches is that the learned knowledge can be easily expressed into plain English and explained to a human user, so facilitating a closer interaction between humans and the machine. Although a well established field since the early '90s [28], logic-based machine learning has traditionally addressed the task of learning knowledge expressible in a very limited form [29] (definite clauses). Our logic-based machine learning systems [2] [7] [21] have extended this field to a wider class of formalisms for knowledge representation, captured by the answer set programming (ASP) semantics [14]. This ASP formalism is truly declarative, and due to its non-monotonicity it is particularly well suited to common-sense reasoning [9, 27, 13]. It allows constructs such as choice rules, hard and weak constraints, and support for default inference and default assumptions. Choice rules and weak constraints are particularly useful for modelling human preferences, as the choice rules can represent the choices available to the user, and the weak constraints can specify which choices a human prefers. The typical workflow in ASP is that a real world problem is encoded as an ASP program, whose *answer sets* – a special subset of the models of the program – correspond to the solutions of the original problem. Because of its expressiveness and efficient solving, ASP is also increasingly gaining attention in industry [10]; for example, in decision support systems [31], in e-tourism [38] and in product configuration [43].

In the recent years we have made fundamental contributions to the field of ILP by extending it to the learning of the full class of ASP programs [33, 40, 36, 7, 21, 25] and this tutorial provides an introduction to these results and to the general field of learning under the answer set semantics. In general, ASP programs can have one, many or even no answer sets. Early approaches to learning ASP programs can mostly be divided into two categories: *brave* learners aim to learn a program such that at least one answer set covers the examples; on the other hand, *cautious* learners aim to find a program which covers the examples in all answer sets. Most of the early ASP-based ILP systems were brave, and several of these are presented in Section 3 of this tutorial. In [21], we showed that some ASP programs cannot be learned using either the brave or the cautious settings, and in fact a combination of *both* brave and cautious semantics is needed. This was the original motivation for the *Learning from Answer Sets* family of frameworks, which we have developed since then and have been shown to be able to learn any ASP program. Section 4 presents these Learning from Answer Sets frameworks and discusses the associated ILASP algorithms. The

generality of the main ASP-based ILP frameworks was investigated, with the aim being to formally define the classes of problems that can be solved by each of these learning frameworks has also been investigated [25]. We re-present and discuss the main results of this investigation in Section 4.

The above is all presented in the context of learning tasks where all examples are assumed to be perfectly labeled, meaning that any inductive solution of a task must cover every example of that task. In practice, of course, examples are unlikely to be perfectly labeled. In real datasets, it is likely that there is *noise*, and a more realistic approach is to search for a hypothesis that covers the majority of examples, and balances the example coverage against the complexity of the hypothesis – dramatically increasing the hypothesis complexity in order to cover a few more examples is undesirable, as these examples may well be incorrectly labeled. We end the tutorial by discussing how ILP frameworks can be extended to learn from noisy examples.

The rest of this document is structured as follows. In the next section we recall the background material necessary for this tutorial. Section 3 covers the early approaches to learning under the answer set semantics. Section 4 introduces the more recent advances including the Learning from Answer Sets frameworks, the generality results for the frameworks and extensions for learning from noisy examples. Much of the material in this tutorial is based on [20].

2 Background

In this section we introduce the background material used in the tutorial.

2.1 Answer Set Programming

Given any atoms $\mathtt{h}, \mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{k}, \mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{n}, \mathtt{c}_1, \ldots, \mathtt{c}_\mathtt{m}$, a rule $\mathtt{h}\,\mathtt{:-}\,\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{n},\ \mathtt{not}\ \mathtt{c}_1, \ldots,\ \mathtt{not}\ \mathtt{c}_\mathtt{m}$ is called a *normal rule*, with $\mathtt{h}$ as the *head* and $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{n},\ \mathtt{not}\ \mathtt{c}_1, \ldots, \mathtt{not}\mathtt{c}_\mathtt{m}$ (collectively) as the *body* ("not" represents negation as failure); a *constraint* is a rule $\mathtt{:-}\,\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{n},\ \mathtt{not}\ \mathtt{c}_1, \ldots,\ \mathtt{not}\ \mathtt{c}_\mathtt{m}$; and a *choice rule* is a rule of the form $\mathtt{l}\{\mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{k}\}\mathtt{u}\,\mathtt{:-}\,\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{n},\ \mathtt{not}\ \mathtt{c}_1, \ldots,\ \mathtt{not}\ \mathtt{c}_\mathtt{m}$ where $\mathtt{l}\{\mathtt{h}_1, \ldots, \mathtt{h}_\mathtt{k}\}\mathtt{u}$ is called an *aggregate*. In an aggregate $\mathtt{l}$ and $\mathtt{u}$ are integers and $\mathtt{h}_\mathtt{i}$, for $\mathtt{1} \leq \mathtt{i} \leq \mathtt{k}$, are atoms. For example, when learning a policy, we may need to learn that in a specific scenario, $\mathtt{sc}_1$, at least one of a set of possible actions, $\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}$, must be executed. This can be expressed in ASP with the following choice rule: $\mathtt{1}\{\mathtt{execute}(\mathtt{a}_1), \ldots, \mathtt{execute}(\mathtt{a}_\mathtt{n})\}\mathtt{n}\,\mathtt{:-}\,\mathtt{holds}(\mathtt{sc}_1)$. This expresses that in a model whenever the scenario $\mathtt{sc}_1$ is true, it must be the case that between 1 and n atoms $\mathtt{execute}(\mathtt{a}_1), \ldots, \mathtt{execute}(\mathtt{a}_\mathtt{n})$ are also true. In other words, whenever the scenario holds, at least one (but possibly more) of the actions must be executed.

A rule R is called *safe* if every variable in R occurs in at least one positive literal in the body of R. For example, the rules $\mathtt{p}(\mathtt{X})\,\mathtt{:-}\,\mathtt{q}(\mathtt{Y}),\ \mathtt{not}\ \mathtt{r}(\mathtt{Y})$ and $\mathtt{p}\,\mathtt{:-}\,\mathtt{q},\ \mathtt{not}\ \mathtt{r}(\mathtt{X})$ are not safe, as X does not occur in any positive literal in their respective body. Unless otherwise specified, an ASP program P is a finite set of safe normal rules, constraints and choice rules.

Given an ASP program P, the Herbrand Base of P, denoted as HB_P, is the set of ground (variable free) atoms that can be formed from the predicates and constants that appear in P. The subsets of HB_P are called the (Herbrand) interpretations of P. Informally, a model of an ASP program P, called *Answer Set* of P, is defined in terms of the notion of *reduct* of P, which is in turn constructed by applying four transformation steps (described below) to the grounding of P. As shown below. a reduct is a definite program. A model of a definite program is an interpretation I that makes every rule in the program true, and a model is *minimal* if it is the smallest such interpretation. Let's see how the reduct of an ASP program is constructed. We said that it uses the grounding of the program, so we can consider just the grounding of a given program P. A ground aggregate $\mathtt{l}\{\mathtt{h_1}, \ldots, \mathtt{h_k}\}\mathtt{u}$ is said to be satisfied by an interpretation I if and only if $\mathtt{l} \leq |I \cap \{\mathtt{h_1}, \ldots, \mathtt{h_k}\}| \leq \mathtt{u}$. As we restrict our programs to sets of normal rules, constraints and choice rules, we use the simplified definitions of the *reduct* for choice rules presented in [23]. Given a program P and an Herbrand interpretation $I \subseteq HB_P$, the reduct P^I is constructed from the grounding of P using the following four transformation steps: from the grounding of P we first remove rules whose bodies contain the negation of an atom in I; secondly, we remove all negative literals from the remaining rules; thirdly, we set $\bot$ (note $\bot \notin HB_P$) to be the head to every constraint, and in every choice rule whose head is not satisfied by I we replace the head with $\bot$; and finally, we replace any remaining choice rule $\mathtt{l}\{\mathtt{h_1}, \ldots, \mathtt{h_m}\}\mathtt{u}\,\mathtt{:\text{-}}\,\mathtt{b_1}, \ldots, \mathtt{b_n}$ with the set of rules $\{\mathtt{h_i}\,\mathtt{:\text{-}}\,\mathtt{b_1}, \ldots, \mathtt{b_n} \mid \mathtt{h_i} \in I \cap \{\mathtt{h_1}, \ldots, \mathtt{h_m}\}\}$. Any $I \subseteq HB_P$ is an *answer set* of P if it is the minimal model of the *reduct* P^I. We denote an answer set of a program P with A and the set of answer sets of P with $AS(P)$. A program P is said to be satisfiable (resp. unsatisfiable) if $AS(P)$ is non-empty (resp. empty).

ASP also allows optimisation over the answer sets according to *weak constraints*. These are rules of the form $:\sim \mathtt{b_1}, \ldots, \mathtt{b_m}, \mathtt{not}\ \mathtt{b_{m+1}}, \ldots, \mathtt{not}\ \mathtt{b_n}.[\mathtt{w@l}, \mathtt{t_1}, \ldots, \mathtt{t_k}]$ where $\mathtt{b_1}, \ldots, \mathtt{b_n}$ are atoms called (collectively) the *body* of the rule, and $\mathtt{w}, \mathtt{l}, \mathtt{t_1} \ldots \mathtt{t_k}$ are all terms with $\mathtt{w}$ called the weight and $\mathtt{l}$ the priority level. We refer to $[\mathtt{w@l}, \mathtt{t_1}, \ldots, \mathtt{t_k}]$ as the *tail* of the weak constraint. A ground instance of a weak constraint W is obtained by replacing all variables in W with ground terms. We assume that all weights and levels of all ground instances of weak constraints are integers. Unlike other ASP rules, *weak constraints* do not affect what is (or is not) an answer set of a program. Instead, they create an ordering $\succ_P$ over $AS(P)$, specifying which answer sets are "preferred" to others. Informally, at each *priority level* $\mathtt{l}$, satisfying weak constraints with level $\mathtt{l}$ means discarding any answer set that does not minimise the sum of the weights of the ground weak constraints (with level $\mathtt{l}$) whose bodies are satisfied. Higher levels are minimised first. For example, the two weak constraints $:\sim \mathtt{mode(L, walk)}, \mathtt{distance(L, D)}.[\mathtt{D@2}, \mathtt{L}]$ and $:\sim \mathtt{cost(L, C)}.[\mathtt{C@1}, \mathtt{L}]$ express a preference ordering over alternative journeys. The first constraint (at priority 2) expresses that the total walking distance (the sum of the distances of journey legs whose mode of transport is $\mathtt{walk}$) should be minimised, and the second constraint expresses that the total cost of the journey should be minimised.

As the first weak constraint has a higher priority level than the second, it is minimised first – so given a journey j_1 with a higher cost than another journey j_2, j_1 is still preferred to j_2 so long as the walking distance of j_1 is lower than that of j_2. The set $ord(P)$ captures the ordering of interpretations induced by P and generalises the $\succ_P$ relation, so it not only includes $\langle A_1, A_2, < \rangle$ if $A_1 \succ_P A_2$, but includes tuples for each binary comparison operator ($<$, $>$, $=$, $\leq$, $\geq$ and $\neq$).

2.2 Inductive Logic Programming

The most common setting for ILP is called *learning from entailment*, where a task consists of a background knowledge B (a pre-defined logic program, defining concepts which may be useful), and two sets of examples (usually atoms) called the positive and negative examples, E^+ and E^-, respectively. The goal is to find a *hypothesis* H such that $\forall e \in E^+$, $B \cup H \models e$ and $\forall e \in E^-$, $B \cup H \not\models e$. When learning definite logic programs, the notion of entailment ($\models$) is usually entailment in the unique minimal Herbrand model of $B \cup H$, but we will see that under the answer set semantics, it is interesting to explore the use of other notions of entailment.

Usually, the search for hypotheses is bounded by a *hypothesis space*, which is the set of all rules allowed to appear in H. In an ILP task, the expressivity of the hypothesis space is defined by a notion of *language bias* of the task. Mode declarations are a popular means of characterising the language bias [30]. They specify which literals may appear in the head and in the body of a hypothesis. Given a language bias the full hypothesis space, also called *search space* and denoted with S_M, is given by the finite set of all the rules that can be constructed according to the given bias. A language bias can be defined as a pair of mode declarations $\langle M_h, M_b \rangle$, where M_h (resp. M_b) are called the *head* (resp. *body*) *mode declarations*. Each mode declaration m_h (resp. m_b) is a literal whose abstracted arguments are either $+\texttt{t}$ or $-\texttt{t}$ or $\#\texttt{t}$, for some constant `t` (referred to as a *type*). Informally, a literal is said to be *compatible* with a mode declaration m if every instance of $+\texttt{t}$ and $-\texttt{t}$ in m has been replaced with a variable, and every $\#\texttt{t}$ with a constant of type `t`. We say that a variable occurs as an *input* (resp. *output*) variable of type `t` if it replaces an argument $+\texttt{t}$ (resp. $-\texttt{t}$). Given a mode bias M, S_M is the set of all rules which are compatible with the mode declarations M.

Definition 1. *Given a set of mode declarations $M = \langle M_h, M_b \rangle$, a normal rule R is in the search space S_M if and only if (i) the head of R is compatible with a mode declaration in M_h; (ii) each body literal of R is compatible with a mode declaration in M_b; (iii) every input variable in the body of R occurs earlier in R, either as an input variable in the head, or an output variable in the body; and (iv) no variable occurs with two different types.*

In the input to most ILP systems a mode declaration is written as atom `class(recall,m)`, where `class` is either `#modeh` or `#modeb`, specifying whether the mode declaration `m` is in M_h or M_b. The *recall* is an optional integer argument, which puts an upper bound on the number of times the mode declaration

can be used in a single rule. In many ILP systems, the types of variables are "enforced" by adding an extra "type" atom to the body for each variable; for instance, for a variable `V` of type `bird`, the atom `bird(V)` is added.

The notion of mode bias given in Definition 1 is commonly used in the ILP literature, but it is not universal. ILASP uses a different notion of mode bias, which we will not present in this tutorial. We refer the reader to `https://www.doc.ic.ac.uk/~ml1909/ILASP/language_biases_2018.pdf` for further details. For simplicity, all the ILASP learning tasks presented in this paper will include an explicit hypothesis space defined in terms of sets of ASP (choice) rules and constraints rather than using a declarative mode bias.

2.3 Complexity Theory

We assume the reader is familiar with the fundamental concepts of complexity, such as Turing machines and reductions; for a detailed explanation, see [34]. $\mathcal{P}$ is the class of all problems which can be solved in polynomial time by a Deterministic Turing Machine (DTM); $\Sigma_0^{\mathcal{P}} = \Pi_0^{\mathcal{P}} = \Delta_0^{\mathcal{P}} = \mathcal{P}$; $\Delta_{k+1}^{\mathcal{P}} = \mathcal{P}^{\Sigma_k^{\mathcal{P}}}$ is the class of all problems which can be solved by a DTM in polynomial time with a $\Sigma_k^{\mathcal{P}}$ oracle. $\Sigma_{k+1}^{\mathcal{P}} = NP^{\Sigma_k^{\mathcal{P}}}$ is instead the class of all problems which can be solved by a non-deterministic Turing Machine in polynomial time with a $\Sigma_k^{\mathcal{P}}$ oracle. Finally. $\Pi_{k+1}^{\mathcal{P}} = \text{co-}NP^{\Sigma_k^{\mathcal{P}}}$ is the class of all problems whose complement can be solved by a non-deterministic Turing Machine in polynomial time with a $\Sigma_k^{\mathcal{P}}$ oracle. $\Sigma_1^{\mathcal{P}}$ and $\Pi_1^{\mathcal{P}}$ are NP and co-NP (respectively). Note, NP is the class of problems which can be solved by a non-deterministic Turing machine in polynomial time and co-NP is the class of problems whose complement is in NP. DP is the class of problems that can be mapped to a pair of problems D_1 and D_2 such that $D_1 \in NP$ and $D_2 \in \text{co-}NP$. It is well known that the following inclusions hold: $\mathcal{P} \subseteq NP \subseteq DP \subseteq \Delta_2^{\mathcal{P}} \subseteq \Sigma_2^{\mathcal{P}}$ and $\mathcal{P} \subseteq \text{co-}NP \subseteq DP \subseteq \Delta_2^{\mathcal{P}} \subseteq \Pi_2^{\mathcal{P}}$ [34].

3 Early Approaches to Logic-based Learning under the Answer Set Semantics

In ASP, there can be one, many or even no answer sets of a program. This leads to two different standard notions of entailment under the answer set semantics: *brave entailment* and *cautious entailment*. Consider, for instance, the following ASP program $P = \{$`1{p,q}1:-r. r. :- not p,r.`$\}$. This program would accept exactly one answer set $A = \{r, p\}$. In this case r and p would be entailed bravely and cautiously. If the first choice rule was instead replaced with `1{p,q}2:-r.`, the program would accept two answer sets $A_1 = \{r, p\}$ and $A_2 = \{r, p, q\}$. In this case r would be cautiously entailed but q would be only bravely entailed. If the additional constraint `:-r` was added to P, then the new program would have no answer set.

These two different notions of entailment naturally lead to two different frameworks for learning from entailment under the answer set semantics: *cautious induction* and *brave induction*. Early approaches to ILP under the answer set semantics tended to adopt cautious induction[1] [16, 42, 39], as this is closer to standard learning from entailment, where examples must be covered in every model. In [41], it was argued that in some cases cautious induction can be too strong as it would require that all positive examples must be true in all answer sets of a given background knowledge and learned hypothesis (this is illustrated in Example 2). In those cases a weaker form of induction – brave induction – is needed. It was in [41] that the notions of *brave* and *cautious* induction were first defined. Brave induction defines an inductive task where all of the examples should be covered in at least one answer set (i.e. entailed under brave entailment in ASP). Note that there should be at least one answer set that covers every example (rather than at least one answer set for each example). Therefore, brave induction cannot specify other brave learning tasks such as enforcing that two examples are both bravely entailed, but not necessarily in the same answer set (as brave induction requires all examples to be covered in the same answer set). In some cases, for instance, we might want to learn a hypothesis that would require to cover some positive example(s) in an answer set and other positive example(s) in other answer sets (of the same learned hypothesis when added to a given background knowledge). These examples would still be bravely entailed but brave induction would not be able to solve tasks requiring such type of coverage. Furthermore, brave induction can only reason about what should be true in at least one answer set of a learned hypothesis (together with the background knowledge). Therefore it cannot reason about what should be true in all answer sets of a program. For this reason, brave induction is incapable of learning constraints.

3.1 Cautious Induction

Cautious induction, first presented in [41], defines a learning task in which all examples should be covered in every answer set (i.e. entailed under cautious entailment in ASP) and $B \cup H$ should be satisfiable (have at least one answer set)[2]. Note that the satisfiability condition is crucial to avoid trivial solutions such as "`:-.`", which eliminate all answer sets.

[1] As the notions had not been defined at the time, they did not call it cautious induction, but the definitions are the same.

[2] The original definitions of brave and cautious induction did not consider atoms which should not be present in an answer set (negative examples). Publicly available algorithms that realise brave induction, on the other hand, do allow for negative examples. We therefore upgrade the definitions in this tutorial to allow negative examples. Note that a negative example `e` can be easily simulated by adding a rule `a:- not e` to the background knowledge and giving `a` as a positive example (where `a` is a new atom that does not appear anywhere in the original task).

Definition 2. *A cautious induction (ILP_c) task T_c is a tuple $\langle B, S_M, \langle E^+, E^- \rangle\rangle$, where B is an ASP program, S_M is a set of ASP rules and E^+ and E^- are sets of ground atoms. A hypothesis $H \subseteq S_M$ is an inductive solution of T_c, written $H \in ILP_c(T)$, if and only if $AS(B \cup H) \neq \emptyset$ and $\forall A \in AS(B \cup H)$, $E^+ \subseteq A$ and $E^- \cap A = \emptyset$.*

Example 1. Consider the ILP_c task $T = \langle B, S_M, \langle E^+, E^- \rangle\rangle$, where:

$$B = \left\{ \begin{array}{l} \texttt{bird(X):-penguin(X).} \\ \texttt{bird(X):-sparrow(X).} \\ \texttt{penguin(b1).} \\ \texttt{sparrow(b2).} \end{array} \right\} \quad S_M = \left\{ \begin{array}{l} \texttt{h}_1 : \texttt{flies(X):-bird(X).} \\ \texttt{h}_2 : \texttt{flies(X):-bird(X),} \\ \qquad\qquad \texttt{not penguin(X).} \\ \texttt{h}_3 : \texttt{0\{flies(X)\}1:-bird(X).} \\ \texttt{h}_4 : \texttt{0\{flies(X)\}1:-bird(X),} \\ \qquad\qquad \texttt{not penguin(X).} \end{array} \right\}$$

$$E^+ = \{\texttt{flies(b2)}\}$$
$$E^- = \{\texttt{flies(b1)}\}$$

The background knowledge B has only one answer set $A = \{\texttt{penguin(b1)}, \texttt{sparrow(b2)}, \texttt{bird(b1)}, \texttt{bird(b2)}\}$.

- $\emptyset \notin ILP_c(T)$ as B has exactly one answer set, and it does not contain flies(b2).
- $\{\texttt{h}_1\} \notin ILP_c(T)$ as $B \cup \{\texttt{h}_1\}$ has exactly one answer set, $A \cup \{\texttt{flies(b1)}\}$, which contains the negative example.
- $\{\texttt{h}_2\} \in ILP_c(T)$ as $B \cup \{\texttt{h}_2\}$ has exactly one answer set, $A \cup \{\texttt{flies(b2)}\}$ which contains the positive example flies(b2) but not the negative example flies(b1).
- $\{\texttt{h}_3\}$ and $\{\texttt{h}_4\}$ are not in $ILP_c(T)$, as they both have answer sets (when combined with B) that do not cover the examples. Specifically, $B \cup \{\texttt{h}_3\}$ has three answer sets: $A_1 = A$, $A_2 = A \cup \{\texttt{flies(b1)}\}$, and $A_3 = A \cup \{\texttt{flies(b2)}\}$. It is clearly not the case that all these three answer sets include the positive example and do not include the negative example. Similarly, $B \cup \{\texttt{h}_4\}$ has two answer sets, $A_1 = A$ and $A_2 = A \cup \{\texttt{flies(b2)}\}$ which also do not all include the positive example.

Limitations of Cautious Induction. Enforcing that examples are covered in every answer set is sometimes too *strong* a requirement, as shown in the following example.

Example 2. Consider the background knowledge $B = \emptyset$ and the hypothesis space $S_M = \{\texttt{h}_1 : \texttt{p:- not q.}\ ;\ \texttt{h}_2 : \texttt{q:- not p.}\}$. There are only two atoms (p and q) in the Herbrand base of $B \cup S_M$. It is impossible to construct an ILP_c task T with background knowledge B and hypothesis space S_M, whatever example from the Herbrand base we consider, that would accept $\{\texttt{h}_1, \texttt{h}_2\}$ as solution and does not accept the empty set as solution. This can be seen as follows. Given that there are only two atoms (p and q) in the Herbrand base of $B \cup S_M$, there are only two atoms which would be meaningful as examples. Neither of them can be given as a positive example as for each atom there is an answer set of $B \cup \{\texttt{h}_1, \texttt{h}_2\}$ that does not contain it. Similarly, neither can be given as a negative example, as

for each atom there is an answer set that contains it. This means that the only ILP_c task T such that $\{\mathtt{h_1}, \mathtt{h_2}\} \in ILP_c(T)$ is $\langle B, S_M, \langle \emptyset, \emptyset \rangle\rangle$. But, clearly for this task, as there are no examples in T, the empty set, $\emptyset$, would be also an inductive solution of T, and it would be the one that, in practice, caution ILP systems would return as they would always search for the shortest possible hypothesis. This means that no examples can be given such that a cautious induction system would return $\{\mathtt{h_1}, \mathtt{h_2}\}$, showing that caution induction is too restrictive.

3.2 Brave Induction

Brave induction (ILP_b) was also formalised in [41]. It defines an inductive task in which all of the examples should be covered in at least one answer set (i.e. entailed under brave entailment in ASP). Note that there should be at least one answer set that covers every example (rather than at least one answer set for each example).

Definition 3. *A brave induction (ILP_b) task T_b is a tuple $\langle B, S_M, \langle E^+, E^- \rangle\rangle$, where B is an ASP program, S_M is a set of ASP rules and E^+ and E^- are sets of ground atoms. A hypothesis $H \subseteq S_M$ is an inductive solution of T_b, written $H \in ILP_b(T)$, if and only if $\exists A \in AS(B \cup H)$ such that $E^+ \subseteq A$ and $E^- \cap A = \emptyset$.*

Example 3. Consider the ILP_b task $T = \langle B, S_M, \langle E^+, E^- \rangle\rangle$, where B, S_M, E^+ and E^- are defined as in Example 2:

$$B = \left\{\begin{array}{l}\mathtt{bird(X)\text{:-}\,penguin(X).}\\ \mathtt{bird(X)\text{:-}\,sparrow(X).}\\ \mathtt{penguin(b1).}\\ \mathtt{sparrow(b2).}\end{array}\right\} \qquad S_M = \left\{\begin{array}{l}\mathtt{h_1 : flies(X)\text{:-}\,bird(X).}\\ \mathtt{h_2 : flies(X)\text{:-}\,bird(X),}\\ \qquad\qquad \mathtt{not\ penguin(X).}\\ \mathtt{h_3 : 0\{flies(X)\}1\text{:-}\,bird(X).}\\ \mathtt{h_4 : 0\{flies(X)\}1\text{:-}\,bird(X),}\\ \qquad\qquad \mathtt{not\ penguin(X).}\end{array}\right\}$$

$$E^+ = \{\mathtt{flies(b2)}\}$$
$$E^- = \{\mathtt{flies(b1)}\}$$

- $\emptyset \notin ILP_b(T)$ as B has exactly one answer set, and it does not contain `flies(b2)`.
- $\{\mathtt{h_1}\} \notin ILP_b(T)$ as $B \cup \{\mathtt{h_1}\}$ has exactly one answer set, and it contains `flies(b1)`.
- $\{\mathtt{h_2}\}, \{\mathtt{h_3}\}, \{\mathtt{h_4}\} \in ILP_b(T)$ as each of $B \cup \{\mathtt{h_2}\}$, $B \cup \{\mathtt{h_3}\}$ and $B \cup \{\mathtt{h_4}\}$ has the answer set $\{\mathtt{penguin(b1), sparrow(b2), bird(b1), bird(b2), flies(b2)}\}$, which contains `flies(b2)` but does not contain `flies(b1)`.

Limitations of Brave Induction Brave induction can only reason about what should be true in at least one answer set of a program. It cannot reason about what should be true in all answer sets of a program. For this reason, brave induction is incapable of learning constraints, as illustrated in the following example. In particular, any solution of an ILP_b task T that includes a constraint is still a solution of T if the constraint is removed, indicating that brave induction omits searching for constraints when learning a solution.

Example 4. Consider the background knowledge $B = \{0\{p\}1.\}$ and a hypothesis space S_M, containing only the constraint `:- p.`, $S_M = \{p.\}$.There is only one atom (`p`) in the Herbrand base of $B \cup S_M$. We show that it is impossible to construct a brave induction task T_b, with background knowledge B and hypothesis space SM , whatever example from the Herbrand base we consider, that accepts {`:- p.`} as solution and does not accept the empty set as solution. This can be seen as follows. Given that there is only one atom (`p`) in the Herbrand base of $B \cup S_M$, there is then one atom which would be meaningful as an example. It must be given as a negative example (as $B \cup$ {`:- p.`} has only one answer set, and it does not contain `p`). But $B \cup \emptyset$ also covers this negative example, as it also has the answer set $\emptyset$, which does not contain `p`. Therefore for any ILP_b task that accepts the constraint {`:- p.`} as a solution, $\emptyset$ is also a solution, meaning that in practice brave induction systems (searching for the shortest hypothesis) will never return the constraint as (part of) a solution.

XHAIL. One of the first logic-based machine learning systems under the answer set programming semantics is the *eXtended Hybrid Abductive Inductive Learning* (XHAIL) [36], which generalises the HAIL [37, 35] algorithm, defined for definite clauses, in order to solve ILP_b tasks for ASP programs with negation as failure. Similarly to HAIL, XHAIL combines abductive and deductive inference. Abductive inference is an *ampliative* form of inference, as it generates knowledge that is not included in the premises of the inference process. Specifically, abduction is the process of reasoning from examples (observations) to possible causes. An abductive inference task takes as input a background knowledge, a set of abducibles (i.e., ground atoms that are not defined in the background knowledge) and set of examples and returns as output possible explanations (i.e. cases in syllogistic terms), also called abductive solutions, that together with the background knowledge, entail (i.e. explain) the examples. It differs from inductive inference in the fact that explanations do not require a process of generalisation during the inference process, whereas induction aims at discovering new general rules from samples (positive and negative) of cases. In other words, abduction is the process of explanation – reasoning from effects to possible causes, whereas induction is the process of generalisation – reasoning from specific cases to general hypothesis.

The XHAIL learning system computes inductive solutions in three phases: an abductive phase; a deductive phase; and an inductive phase. The abductive step takes as abducibles ground atoms that conform with the `modeh` of the language bias of the given task and computes as abductive solution a set of abducible atoms, Δ, such that $B \cup \Delta \models_b (\bigwedge E^+) \wedge (\bigwedge\{\texttt{ not e} \mid \texttt{e} \in E^-\})$. An abductive solution becomes the heads of (ground instances of) rules in the final hypothesis. Next, in the deductive phase, XHAIL computes the set of all ground literals that could go in the body of the rules in the hypothesis. Each of these body atoms `b` is such that $B \cup \Delta \models_b$ `b` and `b` is a ground instance of an atom that conforms to at least one `modeb` declaration. The sets of ground rules with the heads from Δ

and with bodies consisting of literals computed in the deductive phase is referred to as the (ground) Kernel Set $\mathcal{K}$.

Example 5. (from [36])
Consider the ILP_b task $T = \langle B, S_M, \langle E^+, E^- \rangle\rangle$, where B, M, E^+ and E^- are as follows:

$$B = \left\{ \begin{array}{l} \texttt{bird(X):-penguin(X).} \\ \texttt{bird(a).} \\ \texttt{bird(b).} \\ \texttt{bird(c).} \\ \texttt{penguin(d).} \end{array} \right\} M = \left\{ \begin{array}{l} \texttt{\#modeh(flies(+bird))} \\ \texttt{\#modeb(penguin(+bird))} \\ \texttt{\#modeb(not penguin(+bird))} \end{array} \right\}$$

$$E^+ = \left\{ \begin{array}{l} \texttt{flies(a),} \\ \texttt{flies(b),} \\ \texttt{flies(c)} \end{array} \right\} \qquad E^- = \left\{ \texttt{flies(d)} \right\}$$

One abductive explanation of the examples is $\Delta = \{$`flies(a)`, `flies(b)`, `flies(c)`$\}$. This leads to the ground Kernel Set:

$$\mathcal{K} = \left\{ \begin{array}{l} \texttt{flies(a):- not penguin(a).} \\ \texttt{flies(b):- not penguin(b).} \\ \texttt{flies(c):- not penguin(c).} \end{array} \right\}$$

The unground Kernel Set that conforms to the mode bias and the declaration of input variables is given by:

$$\mathcal{K} = \left\{ \texttt{flies(X):- not penguin(X).} \right\}$$

Note that although there are other potential body literals that are entailed by $B \cup \Delta$ and that are declared to be mode body predicates (i.e., `penguin(d)`) they are not added to the ground Kernel Set as they do not form part of a ground instance of a rule that conforms to the mode declarations. According to the declaration of input variables in the language bias body literals need to have the same variable as the one that appears in the head of the rule. The ground literal, `penguin(d)`, for instance, cannot be added to any of the three ground rules of the ground Kernel Set. This is because the unground version of the Kernel Set would give an unground rule with `penguin(Y)` in the body, which violates the input variable constraint of the mode body declaration: an input variable in a mode body predicate must either appear in the head predicate of the rule, or appear as output variable in another body atom. In the mode declaration M given above, variables are only input variables, so any variable that appears in a body predicate must appear in the head predicate of the rule.

The final step of the XHAIL algorithm – the *inductive* step – computes a hypothesis H that conforms to the mode declarations, subsumes the unground Kernel Set and bravely entails the examples. This phase is also supported by an abductive inference process that takes as background knowledge a “transformed”

unground Kernel Set, as abducibles ground instances of a predicate `use`, and as observation the same of examples used in the first phase. The abductive answer determines the literals in the body of the rules of the Kernel Set that need to be maintained in order for the examples to be covered.

Example 6. Consider again the unground Kernel Set computed in Example 5. This is transformed in the following ASP program:

$$\left\{\begin{array}{l}\texttt{flies(X):-use(1,0),try(1,1,X).}\\ \texttt{try(1,1,X):-bird(X), not use(1,1).}\\ \texttt{try(1,1,X):-bird(X),use(1,1), not penguin(X).}\end{array}\right\}$$

The first and second arguments of each of the meta-level atoms `use` and `try` indicate respectively a unique identifier for the object-level rules (starting from 1) in the unground kernel Set, and a unique identifier for the literal in each of these rules (starting from 0 as identifier of the head atom). So, `use(1,0)` means that the head atom `flies(X)` is being *used* (i.e. it is in the hypothesis). The `try` atoms are for testing whether the rule body is satisfied. If the head is being used, then `flies(X)` is true in two cases: (1), the literal `not penguin(X)` is not in the hypothesis (indicated by the first `try` rule); or (2), `not penguin(X)` is true (represented by the second `try` rule).

The transformed Kernel Set is augmented with the choice rule `0{use(1,0), use(1,1)}2`. This phase computes an abductive solution. In the above example, XHAIL uses an ASP solver to compute the smallest abductive answer using the transformed Kernel Set, and this answer gives then hypothesis that subsumes the Kernel Set, conforms to the mode declarations and bravely entails the examples. In the above example, the abductive solution generated during the inductive phase would be $\Delta_i = \{\texttt{use(1,1)}\}$, which indicates that in the final hypothesis, constituted just by the first rule, the first body literal will have to be kept in order to cover the examples correctly.

One major difference between HAIL and XHAIL is that HAIL uses a cover loop approach, whereas XHAIL does not. This is due to the nonmonotonicity of negation as failure: in a cover loop approach, examples that were covered in previous iterations of the cover loop may not be covered in future iterations.

As in general there are many possible abductive solutions Δ, and not all Δ's lead to inductive solutions, XHAIL employs an iterative deepening approach, ordering the Δ's by size and terminating after processing the shortest Δ that leads to a solution. In general, this may not lead to the optimal solution being found, as there may be a large Δ that leads to a shorter hypothesis (e.g. with more individual rules, but fewer overall literals).

INSPIRE. Inspire [18], is an ILP system based on XHAIL, but with some modifications to aid scalability. The main modification is that some rules are "pruned" from the Kernel Set before the XHAIL's inductive phase. Both XHAIL and Inspire use a meta-level ASP program to perform the inductive phase, and the ground Kernel Set is generalised into a first order Kernel Set (using the mode

declarations to determine which arguments of which predicates should become variables). Inspire prunes rules that have fewer than Pr instances in the ground Kernel Set (where Pr is a parameter of Inspire). The intuition is that if a rule is necessary to cover many examples then it is likely to have many ground instances in the Kernel Set. Clearly this is an approximation, so Inspire is not guaranteed to find the optimal hypothesis in the inductive phase. In fact, as XHAIL is not guaranteed to find the optimal inductive solution of the task (as it may pick the "wrong" abductive solution), this means that Inspire may be even further from computing optimal solutions.

ILED. ILED [17] is an incremental algorithm, based on XHAIL. It is targeted at learning Event Calculus [19] theories and, therefore, its examples are slightly different in that they are grouped into *time windows*. The examples are processed one at a time and at each timepoint the hypothesis is revised so that it covers all examples in all windows that have been processed so far.

ILED has been shown to be much more scalable than XHAIL when processing large numbers of examples divided into time windows [17]. On the other hand, like XHAIL, ILED is not guaranteed to find the optimal solution of a task. In fact, this incompleteness with respect to optimal solutions is more severe in ILED than in XHAIL, as it can also occur because of the incremental nature of the algorithm. Although at each step the revision may be optimal, the combination of every revision may result in a longer hypothesis than could have been found if all examples had been processed together.

ASPAL. The algorithms presented so far follow a bottom-up approach for searching for solutions within a given hypothesis space specified by a language bias. In the cases of XHAIL, ILED and INSPIRE the algorithms compute first a most specific (set of) clauses that cover the examples, which constitute the "bottom element" of the search space, and then try to generalise it searching for more general solutions within the search space. But alternative approaches to the search for inductive solutions have been proposed in the literature. These are referred to as meta-level approaches with top-down search. An example of such algorithms is the Top-directed Abductive Learning (TAL) system [6]. This system solves an ILP task by automatically translating it into a semantically equivalent abductive inference task, whose background knowledge is given by the background knowledge of the learning task augmented with *meta-level* representation of the hypothesis space, and the observation to explain is given, as in XHAIL, by the conjunction of the examples. The inference process performs a top-down abductive search [32] for abductive solutions that explain the observation. Such an abductive solution is then translated back into rules that are guaranteed to correspond to an inductive solution of the original brave inductive task. The transformation relies upon a one-to-one mapping that translates each (normal) rule of the hypothesis space into a meta-level representation and uses a "meta-program", called *top theory*, that captures possible ways of constructing such rules in terms of their meta-level representation. The main advantage

of this approach is its ability to solve ILP tasks that require recursive rules as solutions or rules that are interdependent (e.g., predicates that appear in the body of a rule can also appear in the head of another rule belonging to the same solution). However, a drawback of this approach is its scalability. The abductive reasoning engine used by TAL is implemented in Prolog. As such, it suffers from redundant inference steps, which causes its computational time to be affected by the size of the hypothesis space and the number of examples of a given ILP task.

The ASPAL [7] algorithm is a brave induction system that aims at addressing the limitations of the TAL system by using an ASP implementation: as in TAL, an ILP task is translated into a meta-level program, but in ASPAL this is an ASP program. Given an ILP_b task $T_b = \langle B, S_M, \langle E^+, E^- \rangle\rangle$, where S_M is defined by a given set of mode declarations M, the first step is to compute a set of *skeleton rules* Sk_M. These are the set of rules R, such that there is an $R' \in S_M$, where each constant in R' is replaced by a placeholder variable in R.

Example 7. Consider the mode declarations M.

$$M = \left\{ \begin{array}{l} \#\texttt{modeh(penguin(+bird))} \\ \#\texttt{modeb(2, not can(+bird,} \#\texttt{ability))} \end{array} \right\}$$

The first argument of the mode body declaration is called the *recall* and it expresses the constraint that this mode declaration can be used at most twice per rule in the hypothesis space. There are three skeleton rules:

$$Sk_M = \left\{ \begin{array}{l} \texttt{penguin(X):-bird(X)} \\ \texttt{penguin(X):-bird(X), not can(X,C1)} \\ \texttt{penguin(X):-bird(X), not can(X,C1), not can(X,C2)} \end{array} \right\}$$

Note that the hypothesis space S_M consists of the rules in Sk_M but where $C1$ and $C2$ have been replaced with constants of type `ability`.

Each skeleton rule $R \in Sk_M$ is associated with a unique meta-level atom $\texttt{rule}(\texttt{R}_{\texttt{id}}, \texttt{C}_1, \ldots, \texttt{C}_\texttt{n})$, denoted as R_{meta}, where $\texttt{C}_1, \ldots, \texttt{C}_\texttt{n}$ are the "constant placeholder" variables in R. For each rule $R' \in S_M$, we similarly write R'_{meta} to denote the ground atom representing R' (where each "constant placeholder" variable has been replaced with a constant of the correct type). Informally, consider for example the second rule `penguin(X):-bird(X), not can(X,C1)` in Sk_M. Its associated meta-level atom would be $rule(2, C1)$. Now assuming, for the sake of the argument, that $ability(fly)$ is true in the background knowledge of the given learning task, the ASPAL computation may given rise for instance to a ground instance $rule(2, fly)$, which would then be used by ASPAL to generate the corresponding rule $R' \in S_M$ given by `penguin(X):-bird(X), not can(X,fly)` in the final inductive solution.

Using this notion of skeleton rules, the ASPAL system automatically constructs an ASP meta-level representation of a learning task by adding to the background theory B of the learning task, the set of skeleton rules, each augmented with an additional body literal given by the associated meta-level atom.

These meta-level atoms are considered to be abducible and their truth is determined by a choice rule which is used by the ASP solver to select the minimal number of such atoms (corresponding to the minimal number of rules $R' \in S_M$) so that the examples of the given learning task are covered. This is captured formally by the following definition.

Definition 4. *Let T be the ILP_b task $\langle B, S_M, \langle \{\mathtt{e}_1^+, \ldots, \mathtt{e}_n^+\}, \{\mathtt{e}_1^-, \ldots, \mathtt{e}_m^-\}\rangle\rangle$, where S_M is characterised by the set of mode declarations M. Let Sk_M be the set of skeleton rules derivable from M. The ASPAL meta-representation is the program consisting of the following components:*

- B
- $\mathtt{h}$:- $\mathtt{b}_1, \ldots \mathtt{b}_{\mathtt{rl}}, \mathtt{R}_{\mathtt{meta}}$, *for each rule $R \in Sk_M$, where R is the rule* $\mathtt{h}$:- $\mathtt{b}_1, \ldots, \mathtt{b}_{\mathtt{rl}}$.
- *A choice rule* $0\{\mathtt{ab}_1, \ldots, \mathtt{ab}_k\}\mathtt{k}$., *where* $\{\mathtt{ab}_1, \ldots, \mathtt{ab}_k\} = \{R_{meta} \mid R \in Sk_M\}$[3]
- *The rule* `goal :-` $\mathtt{e}_1^+, \ldots, \mathtt{e}_n^+$, `not` $\mathtt{e}_1^-, \ldots,$ `not` $\mathtt{e}_m^-$.
- *The constraint* `:- not goal`.

We refer to the answer sets of this meta representation as meta-level answer sets, and the answer sets of $B \cup H$ as object-level answer sets. Each meta-level answer set A represents a single hypothesis H (defined by the `rule` atoms in A). Each meta-level answer set also contains exactly one object-level answer set of $B \cup H$ that contains all of the positive examples and none of the negative examples (enforced by the `goal` constraint).

Example 8. Consider the ILP_b task $T = \langle B, S_M, E^+, E^- \rangle$, where S_M is characterised by the mode declarations in Example 7.

$$B = \left\{ \begin{array}{l} \mathtt{bird(a).} \\ \mathtt{bird(b).} \\ \mathtt{can(a, fly).} \\ \mathtt{can(b, swim).} \\ \mathtt{ability(fly).} \\ \mathtt{ability(swim).} \end{array} \right\} E^+ = \{\mathtt{penguin(b)}\} \; E^- = \{\mathtt{penguin(a)}\}$$

The ASPAL meta-level representation is shown in Figure 1. Note that each skeleton rule R has been appended with the atom R_{meta}, where each of the arguments other than the identifier R_{id} is a variable representing a placeholder for a constant. The choice rule on the other hand contains atoms R'_{meta}, generated by instantiating the constant placeholder variable with all possible constant values allowed by the constant type. In this way each ground R' is an instance of a skeleton rule R. The answer sets of this program can be mapped to the ILP_b inductive solutions of the task T_b. For example, the answer set {`bird(a)`,

[3] This is a slight simplification. In the ASPAL algorithm, this is a choice rule using conditional literals, in order to delegate the grounding of the possible constants to the ASP solver. The ground version of ASPAL's choice rule is identical to the one presented in this definition.

```
bird(a).
bird(b).
can(a, fly).
can(b, swim).
ability(fly).
ability(swim).

penguin(X) :- bird(X), rule(1).
penguin(X) :- bird(X), not can(X, C1), rule(2, C1).
penguin(X) :- bird(X), not can(X, C1), not can(X, C2), rule(3, C1, C2).

0{rule(1), rule(2, fly), rule(2, swim), rule(3, fly, swim)}4.

goal :- penguin(b), not penguin(a).
:- not goal.
```

Fig. 1. The ASPAL meta-level representation for the learning task in Example 8.

`bird(b)`, `can(a, fly)`, `can(b, swim)`, `ability(fly)`, `ability(swim)`, `penguin(b)`, `rule(2, fly)`, `goal`} shows that the hypothesis
{`penguin(X) :- bird(X), not can(X, fly).`} is a solution of the ILP_b task T.

In the ASPAL algorithm, this meta representation is combined with an optimisation statement (similar to weak constraints in ASP), which orders the meta-level answer sets by the length of the hypothesis that they represent. This optimisation statement is equivalent to adding a weak constraint $:\sim \mathtt{R_{meta}}.[|\mathtt{R}|@1, \mathtt{R_{meta}}]$ for each R in Sk_M, which means that the total penalty paid by a meta-level answer set at priority level 1 is the length of the hypothesis generated from the answer set. Note that when computing $|R|$ the "type" atoms in R (such as `bird(X)` in the rule above) are not counted.

ASPAL has been proven to be sound and complete with respect to the optimal inductive solutions of any brave induction task [5]. This means that, unlike XHAIL and XHAIL-based algorithms, ASPAL is guaranteed to return an optimal inductive solution of any brave induction task (resources permitting).

RASPAL. ASPAL scales poorly with respect to the size of $ground(B \cup S_M)$ [2]. One of the main factors in the size of this ground program is the number of body literals that are allowed to appear in a rule in the hypothesis space. RASPAL [2] addresses this limitation by iteratively refining a hypothesis until all of the examples in an ILP_b task are covered. At each step, the number of literals that are allowed to be added to the hypothesis is restricted, meaning that the grounding in RASPAL is often significantly smaller than the meta-level program in ASPAL. In [1] it was shown that RASPAL significantly outperforms ASPAL on some learning tasks with large problem domains and large hypothesis spaces.

3.3 Induction of Stable Models.

Induction of stable models [33] (ILP_{sm}), generalises ILP_b, in order to allow conditions to be set over multiple answer sets. The examples of an ILP_{sm} task are *partial interpretations.*

Definition 5. *A* partial interpretation *e is a pair of sets of atoms $\langle e^{inc}, e^{exc}\rangle$. We refer to e^{inc} and e^{exc} as the* inclusions *and* exclusions *respectively. An interpretation I is said to* extend *e if and only if $e^{inc} \subseteq I$ and $e^{exc} \cap I = \emptyset$.*

Example 9. Consider the partial interpretation $e = \langle\{\texttt{p, q}\}, \{\texttt{r, s}\}\rangle$.

- $\{\texttt{p}\}$ does not extend e, as it does not contain q.
- $\{\texttt{p, q, r}\}$ does not extend e, as it contains r.
- $\{\texttt{p, q}\}$ extends e, as it contains all of e's inclusions, and none of e's exclusions.
- $\{\texttt{p, q, t}\}$ extends e, as it contains all of e's inclusions, and none of e's exclusions.

Induction of stable models is formalised in Definition 6.

Definition 6. *An induction of stable models (ILP_{sm}) task T_{sm} is a tuple $\langle B$, S_M, $\langle E\rangle\rangle$, where B is an ASP program, S_M is the hypothesis space and E is a set of example partial interpretations. A hypothesis H is an inductive solution of T_{sm} if and only if $H \subseteq S_M$ and $\forall e \in E$, $\exists A \in AS(B \cup H)$ such that A extends e.*

Note that a brave induction task can be thought of as a special case of induction of stable models (with $|E| = 1$ and the inclusions and exclusions of the only partial interpretation example being the positive and negative examples of the brave task, respectively).

Example 10. Consider the ILP_{sm} task $T = \langle B, S_M, \langle E\rangle\rangle$, where:

$$B = \emptyset \qquad\qquad E = \left\{ \begin{array}{l} \langle\{\texttt{p}\}, \{\texttt{q}\}\rangle, \\ \langle\{\texttt{q}\}, \{\texttt{p}\}\rangle \end{array} \right\}$$

$$S_M = \left\{ \begin{array}{l} \texttt{h}_1 : \texttt{p :- not q.} \\ \texttt{h}_2 : \texttt{q :- not p.} \end{array} \right\}$$

$\{\texttt{h}_1, \texttt{h}_2\}$ is the only subset of the hypothesis space that is an inductive solution of T, as it is the only hypothesis that has answer sets that extend both of the examples.

Note that, although induction of stable models is a generalisation of brave induction, it is still incapable of learning constraints. This is because, similarly to brave induction, it can only give examples of what should be (in) an answer set, rather than examples of what should not be an answer set.

4 Learning from Answer Sets and ILASP

In the previous section, we presented the main frameworks for learning ASP programs, which fall into two categories: either the examples must be covered in at least one answer set of the learned program (brave induction [41] and induction of stable models [33]), or the examples must be covered in every answer set of the learned program (cautious induction [41]). Work on using brave induction (such as [36] and [7]) has often only considered learning stratified programs[4]. In general, however, ASP programs can have one, many or even no answer sets. Example 11 presents a program H describing the rules of Sudoku, and shows that no brave induction, induction of stable models or cautious induction task could possibly have H as an optimal solution.

Example 11. Consider a background knowledge B that contains definitions of the structure of a 4x4 Sudoku grid; i.e. definitions of `cell`, `same_row`, `same_col` and `same_block` (where `same_row`, `same_col` and `same_block` are true only for two *different* cells in the same row, column or block).

$$B = \left\{\begin{array}{l}
\texttt{cell((1,1)).} \quad \texttt{cell((1,2)).} \quad \dots \quad \texttt{cell((4,4)).} \\
\texttt{same_row((X1,Y),(X2,Y)):-cell((X1,Y)),cell((X2,Y)),X1} \neq \texttt{X2.} \\
\texttt{same_col((X,Y1),(X,Y2)):-cell((X,Y1)),cell((X,Y2)),Y1} \neq \texttt{Y2.} \\
\texttt{block((1,1),1).} \quad \texttt{block((1,2),1).} \quad \texttt{block((2,1),1).} \quad \texttt{block((2,2),1).} \\
\texttt{block((3,1),2).} \quad \texttt{block((3,2),2).} \quad \texttt{block((4,1),2).} \quad \texttt{block((4,2),2).} \\
\texttt{block((1,3),3).} \quad \texttt{block((1,4),3).} \quad \texttt{block((2,3),3).} \quad \texttt{block((2,4),3).} \\
\texttt{block((3,3),4).} \quad \texttt{block((3,4),4).} \quad \texttt{block((4,3),4).} \quad \texttt{block((4,4),4).} \\
\texttt{same_block(C1,C2):-block(C1,B),block(C2,B),C1} \neq \texttt{C2.}
\end{array}\right\}$$

One hypothesis H that describes the correct rules of Sudoku is as follows:

$$H = \left\{\begin{array}{l}
\texttt{1\{value(C,1),value(C,2),value(C,3),value(C,4)\}1:-cell(C).} \\
\texttt{:-same_row(C1,C2),value(C1,V),value(C2,V).} \\
\texttt{:-same_col(C1,C2),value(C1,V),value(C2,V).} \\
\texttt{:-same_block(C1,C2),value(C1,V),value(C2,V).}
\end{array}\right\}$$

Let S_M be a set of rules which contains the rules in H (for the purposes of this example, it does not matter which other rules it contains). There is no ILP_b, ILP_{sm} or ILP_c task such that H is a solution, and no subset of H is a solution. In practice, as ILP systems tend to search for a solution that is as short as possible (called an optimal solution), no system for ILP_b, ILP_{sm} or ILP_c will return H as the solution. We now show that no task exists, for any of the three frameworks, for which H is an optimal solution.

- Assume that there is an ILP_b task T_b with background knowledge B such that H is a solution of T_b. Then there must be at least one answer set of $B \cup H$ that contains all of the positive examples of T_b and none of the

[4] Both XHAIL [36] and ASPAL [7] support learning non-stratified programs, but the background knowledge and hypothesis space of each of the example tasks in [36] and [7] is stratified.

negative examples of T_b. But this answer set must also be an answer set of $B \cup \{\texttt{1}\{\texttt{value(C, 1)}, \texttt{value(C, 2)}, \texttt{value(C, 3)}, \texttt{value(C, 4)}\}\texttt{1 :- cell(C).}\}$, as the constraints in H only rule out answer sets. Hence, $H' = \{\texttt{1}\{\texttt{value(C, 1)}, \texttt{value(C, 2)}, \texttt{value(C, 3)}, \texttt{value(C, 4)}\}\texttt{1 :- cell(C).}\}$ must also be an inductive solution of T_b. As H' is shorter than H, this means that H cannot possibly be an optimal solution of T_b.

- The argument for ILP_{sm} is similar to ILP_b. Assume there is an ILP_{sm} task T_{sm} with background knowledge B such that H is a solution of T_{sm}. Then for each example e, there must be at least one answer set A_e of $B \cup H$, such that A_e extends e. In each case, A_e must also be an answer set of $B \cup \{\texttt{1}\{\texttt{value(C, 1)}, \texttt{value(C, 2)}, \texttt{value(C, 3)}, \texttt{value(C, 4)}\}\texttt{1 :- cell(C).}\}$, as the constraints in H only rule out answer sets. Hence, the hypothesis $H' = \{\texttt{1}\{\texttt{value(C, 1)}, \texttt{value(C, 2)}, \texttt{value(C, 3)}, \texttt{value(C, 4)}\}\texttt{1 :- cell(C).}\}$ must also be an inductive solution of T_{sm}. As H' is shorter than H, this means that H cannot possibly be an optimal solution of T_{sm}.
- If we use ILP_c to learn H, we have to give examples which are either true in every answer set of $B \cup H$, or false in every answer set. Therefore, we could not give any meaningful examples about the `value` predicate – for each atom $\texttt{value(x, y)}$ (where `x` and `y` range from 1 to 4), there is at least one answer set of $B \cup H$ that contains $\texttt{value(x, y)}$ and at least one that does not; this means that if $\texttt{value(x, y)}$ is given as either a positive or negative example, H will not be a solution of the task. This means that for any ILP_c task $T_c = \langle B, S_M, E^+, E^- \rangle$ such that H is a solution, $E^+ \subseteq \{\texttt{a} \mid \forall A \in AS(B), \texttt{a} \in A\}$ and $E^- \subseteq \{\texttt{a} \mid \forall A \in AS(B), \texttt{a} \notin A\}$. Hence, for any such task, $\emptyset$ must be a solution of T_c, meaning that H cannot be an optimal solution.

The problem with using either brave or cautious induction to learn general ASP programs is that brave induction can only reason about what should be true in at least one answer set of the learned program, which can be far too weak a condition, and cautious induction can only express what should be true in all answer sets of a program, which can be far too strong a condition. Furthermore, examples in both frameworks are atoms. In ASP it is common [9] to represent a problem such that the answer sets are solutions (see Figure 2 (a)). In order to learn ASP programs, examples should therefore be of what should (or should not) be an answer set of the program (Figure 2 (b)). In the context of learning the rules of Sudoku using the representation in Example 11, this corresponds to giving examples of Sudoku grids rather than the values of individual cells.

In practice, there may be some atoms whose values are unknown before learning. It is therefore more practical to consider learning from partial interpretations rather than full interpretations. This setting, under the answer set semantics, is the basis of the *Learning from Answer Sets* framework.

A learning from answer sets task consists of an ASP background knowledge B, a hypothesis space and sets of positive and negative partial interpretation examples. The goal is to find a hypothesis H that has at least one answer set (when combined with B) that extends each positive example, and no answer set

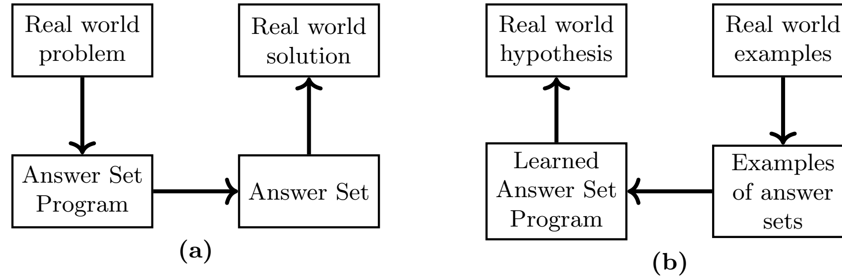


Fig. 2. (a) shows the general paradigm of answer set programming [4]; (b) shows the general idea of Learning from Answer Sets.

that extends any negative examples. Note that each positive example could be extended by a different answer set of the learned program.

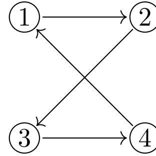
Definition 7. *A* Learning from Answer Sets *(ILP_{LAS}) task is a tuple $T = \langle B, S_M, \langle E^+, E^- \rangle\rangle$ where B is an ASP program, S_M a set of ASP rules and E^+ and E^- are finite sets of partial interpretations. A hypothesis $H \subseteq S_M$ is an* inductive solution *of T if and only if:*

1. $\forall e^+ \in E^+ \; \exists A \in AS(B \cup H)$ *such that A extends e^+*
2. $\forall e^- \in E^- \; \nexists A \in AS(B \cup H)$ *such that A extends e^-*

Example 12. Consider the problem of learning the definition of what it means for a graph to be Hamiltonian.[5] The background knowledge B defines what it means to be a graph, up to size 4.

$$B = \left\{ \begin{array}{l} \texttt{1\{size(1), size(2), size(3), size(4)\}1.} \\ \texttt{node(1..S) :- size(S).} \\ \texttt{0\{edge(N1, N2)\}1 :- node(N1), node(N2).} \end{array} \right\}$$

The answer sets of B exactly represent the graphs of size 1 to 4. For example, the answer set {`size(4)`, `node(1)`, `node(2)`, `node(3)`, `node(4)`, `edge(1, 2)`, `edge(2, 3)`, `edge(3, 4)`, `edge(4, 1)`} represents the graph G:



The program H can be used to determine whether a graph is Hamiltonian or not. The answer sets of $B \cup H$ correspond exactly to the Hamiltonian graphs of size 1 to 4.

[5] A graph is Hamiltonian if it contains a cycle that visits each node exactly once.

$$H = \left\{ \begin{array}{l} \texttt{0\{in(V0,V1)\}1 :- edge(V0,V1).} \\ \texttt{reach(V0) :- in(1,V0).} \\ \texttt{reach(V1) :- in(V0,V1), reach(V0).} \\ \texttt{:- node(V0), not reach(V0).} \\ \texttt{:- in(V0,V1), in(V0,V2), V1! = V2.} \end{array} \right\}$$

The graph G can be represented as a partial interpretation $\langle\{\texttt{size(4)}, \texttt{edge(1,2)}, \texttt{edge(2,3)}, \texttt{edge(3,4)}, \texttt{edge(4,1)}\}, \{\texttt{edge(1,1)}, \texttt{edge(1,3)}, \texttt{edge(1,4)}, \texttt{edge(2,1)}, \texttt{edge(2,2)}, \texttt{edge(2,4)}, \texttt{edge(3,1)}, \texttt{edge(3,2)}, \texttt{edge(3,3)}, \texttt{edge(4,2)}, \texttt{edge(4,3)}, \texttt{edge(4,4)}\}\rangle$.

Given sufficient positive and negative examples of Hamilton graphs, it is possible to learn the hypothesis H using the ILASP system for solving ILP_{LAS} tasks. Similarly to the Sudoku program in Example 11, it is impossible to learn H with any of the previous frameworks.

Since the original ILP_{LAS} framework was introduced in [21], it has been extended in several ways. The rest of this section presents each of these extensions.

4.1 Preference Learning in ASP

Preference Learning has received much attention over the last decade from within the machine learning community. A popular approach to preference learning is *learning to rank* [11, 12], where the goal is to learn to rank any two objects given some examples of pairwise preferences (indicating that one object is preferred to another). While in previous work ILP systems such as TILDE [3] and Aleph [44] have been applied to preference learning [8, 15], this has addressed learning ratings, such as `good`, `poor` and `bad`, rather than rankings over the examples. Ratings are not expressive enough if we want to find an optimal solution as we may rate many objects as `good` when some are better than others. ASP, on the other hand, allows the expression of preferences through *weak constraints*.

Weak constraints do not affect what is, or is not, an answer set of a program. Instead, they create a preference ordering over the answer sets of a program; i.e. they allow us to specify which answer sets are preferred to other answer sets. Example 13 shows how a set of preferences can be encoded as weak constraints.

Example 13. Consider the problem of using a user's preferences over alternative journeys, in order to select the optimal journey. Let A, B, C and D be the journeys represented by the following sets of attributes. Each journey is split into a number of *legs*, in which a single mode of transport is used.

$$\left\{ \begin{array}{l} \texttt{leg_mode(1,walk),} \\ \texttt{leg_crime_rating(1,2),} \\ \texttt{leg_distance(1,500),} \\ \texttt{leg_mode(2,bus),} \\ \texttt{leg_crime_rating(2,4),} \\ \texttt{leg_distance(2,3000)} \end{array} \right\} \quad \text{(A)}$$

$$\left\{\begin{array}{l}\texttt{leg_mode}(1,\texttt{bus}),\\ \texttt{leg_crime_rating}(1,2),\\ \texttt{leg_distance}(1,4000),\\ \texttt{leg_mode}(2,\texttt{walk}),\\ \texttt{leg_crime_rating}(2,5),\\ \texttt{leg_distance}(2,1000)\end{array}\right\}$$

(B)

(C)

$$\left\{\begin{array}{l}\texttt{leg_mode}(1,\texttt{bus}),\\ \texttt{leg_crime_rating}(1,2),\\ \texttt{leg_distance}(1,400),\\ \texttt{leg_mode}(2,\texttt{bus}),\\ \texttt{leg_crime_rating}(2,4),\\ \texttt{leg_distance}(2,3000)\end{array}\right\}$$

$$\left\{\begin{array}{l}\texttt{leg_mode}(1,\texttt{bus}),\\ \texttt{leg_crime_rating}(1,5),\\ \texttt{leg_distance}(1,2000),\\ \texttt{leg_mode}(2,\texttt{walk}),\\ \texttt{leg_crime_rating}(2,1),\\ \texttt{leg_distance}(2,2000)\end{array}\right\}$$

(D)

The following weak constraints H give a preference ordering to the journeys A to D.

$$H = \left\{\begin{array}{l}:\sim \texttt{leg_mode}(\texttt{L},\texttt{walk}),\texttt{leg_crime_rating}(\texttt{L},\texttt{C}),\texttt{C} > 4.[1@3,\texttt{L},\texttt{C}]\\ :\sim \texttt{leg_mode}(\texttt{L},\texttt{bus}).[1@2,\texttt{L}]\\ :\sim \texttt{leg_mode}(\texttt{L},\texttt{walk}),\texttt{leg_distance}(\texttt{L},\texttt{D}).[\texttt{D}@1,\texttt{L},\texttt{D}]\end{array}\right\}$$

The first weak constraint in H means that the user would like to avoid walking through an area with a crime rating higher than 4. A journey pays a penalty of 1 at priority level 3 for each leg of the journey that involves walking though such an area. As there is no weak constraint in H with a priority level higher than 3, this preference is the most important. The second weak constraint (at priority level 2) means that the user would like to take as few buses as possible. The third weak constraint (at priority level 1) means that the user would like to minimise the distance that they have to walk. Note that, as a penalty of the distance is paid for each leg where the user has to walk, the total penalty is equal to the total walking distance of the journey. Given these preferences, A is the best journey, followed by D, then C and then B.

The hypothesis in Example 13 could be learned by giving examples of which journeys are preferred to which other journeys. For the preferences to be learned as weak constraints, this would require examples of pairs of answer sets, such that the first is preferred to the second. In fact, each ordering example contains two partial interpretations, rather than two complete answer sets. Examples can also be given with any of the operators $<$, $\leq$, $=$, $\neq$, $>$ or $\geq$. The $<$ operator, for example, indicates that the first partial interpretation is preferred to the second; whereas the $=$ operator specifies that the two partial interpretations are equal.

Definition 8. *An* ordering example *is a tuple $o = \langle e_1, e_2, op\rangle$ where e_1 and e_2 are partial interpretations and op is a binary comparison operator ($<$, $>$, $=$, $\leq$, $\geq$ or $\neq$).*

As ordering examples contain two partial interpretations, rather than two full interpretations, there are two possible semantics to give to the examples. The

brave semantics indicates that there should be at least one pair of answer sets extending the pair of partial interpretations, which are ordered according to the operator. The *cautious* semantics, on the other hand, indicates that every pair of answer sets that extend the pair of partial interpretations should be ordered according to the operator.

Definition 9. *Let* $o = \langle e_1, e_2, op\rangle$ *be an ordering example. An ASP program* P bravely respects o *iff* $\exists A_1, A_2 \in AS(P)$ *such that all of the following conditions hold: (i)* A_1 *extends* e_1*; (ii)* A_2 *extends* e_2*; and (iii)* $\langle A_1, A_2, op\rangle \in ord(P)$*.* P cautiously respects o *iff* $\nexists A_1, A_2 \in AS(P)$ *such that all of the following conditions hold: (i)* A_1 *extends* e_1*; (ii)* A_2 *extends* e_2*; and (iii)* $\langle A_1, A_2, op\rangle \notin ord(P)$*.*

Definition 10 defines the notion of *Learning from Ordered Answer Sets* (ILP_{LOAS}).

Definition 10. *A* Learning from Ordered Answer Sets *task is a tuple* $T = \langle B, S_M, \langle E^+, E^-, O^b, O^c\rangle\rangle$ *where* B *is an ASP program,* S_M *is a set of ASP rules,* E^+ *and* E^- *are finite sets of partial interpretations and* O^b *and* O^c *are finite sets of ordering examples over* E^+ *called brave and cautious orderings. A hypothesis* $H \subseteq S_M$ *is an inductive solution of* T *if and only if:*

1. $H \in ILP_{LAS}(\langle B, S_M, \langle E^+, E^-\rangle\rangle)$
2. $\forall o \in O^b$ $B \cup H$ *bravely respects* o
3. $\forall o \in O^c$ $B \cup H$ *cautiously respects* o

Note that the orderings are only over positive examples. The justification behind this restriction is that there does not appear to be any scenario where a hypothesis would need to respect an ordering of a pair of partial interpretations that are not extended by any pair of answer sets of $B \cup H$.

Example 14. Recall the journey preferences in Example 13. Consider the background knowedge B, which defines a set of possible journeys.

$$B = \left\{ \begin{array}{l} \texttt{1\{leg(1),...,leg(5)\}5.} \\ \texttt{1\{leg_mode(L,walk),leg_mode(L,bus)\}1:-leg(L).} \\ \texttt{1\{leg_crime_rating(L,1),...,leg_crime_rating(L,4000)\}1:-leg(L).} \\ \texttt{1\{leg_distance(L,0),...,leg_distance(L,4000)\}1:-leg(L).} \end{array} \right\}$$

Journeys A to D of Example 13 can be represented by the four positive examples e_A to e_D.

$$\left\langle \left\{ \begin{array}{l} \texttt{leg_mode(1,walk),} \\ \texttt{leg_crime_rating(1,2),} \\ \texttt{leg_distance(1,500),} \\ \texttt{leg_mode(2,bus),} \\ \texttt{leg_crime_rating(2,4),} \\ \texttt{leg_distance(2,3000)} \end{array} \right\}, \left\{ \begin{array}{l} \texttt{leg(3),} \\ \texttt{leg(4),} \\ \texttt{leg(5)} \end{array} \right\} \right\rangle$$

(e_A)

$$\left\langle \left\{ \begin{array}{l} \texttt{leg_mode}(1, \texttt{bus}), \\ \texttt{leg_crime_rating}(1, 2), \\ \texttt{leg_distance}(1, 400), \\ \texttt{leg_mode}(2, \texttt{bus}), \\ \texttt{leg_crime_rating}(2, 4), \\ \texttt{leg_distance}(2, 3000) \end{array} \right\}, \left\{ \begin{array}{l} \texttt{leg}(3), \\ \texttt{leg}(4), \\ \texttt{leg}(5) \end{array} \right\} \right\rangle \quad (e_C)$$

$$\left\langle \left\{ \begin{array}{l} \texttt{leg_mode}(1, \texttt{bus}), \\ \texttt{leg_crime_rating}(1, 2), \\ \texttt{leg_distance}(1, 4000), \\ \texttt{leg_mode}(2, \texttt{walk}), \\ \texttt{leg_crime_rating}(2, 5), \\ \texttt{leg_distance}(2, 1000) \end{array} \right\}, \left\{ \begin{array}{l} \texttt{leg}(3), \\ \texttt{leg}(4), \\ \texttt{leg}(5) \end{array} \right\} \right\rangle \quad (e_B)$$

$$\left\langle \left\{ \begin{array}{l} \texttt{leg_mode}(1, \texttt{bus}), \\ \texttt{leg_crime_rating}(1, 5), \\ \texttt{leg_distance}(1, 2000), \\ \texttt{leg_mode}(2, \texttt{walk}), \\ \texttt{leg_crime_rating}(2, 1), \\ \texttt{leg_distance}(2, 2000) \end{array} \right\}, \left\{ \begin{array}{l} \texttt{leg}(3), \\ \texttt{leg}(4), \\ \texttt{leg}(5) \end{array} \right\} \right\rangle \quad (e_D)$$

As these positive examples completely represent each journey, there is exactly one answer set of B that extends each example. Therefore there is no distinction between brave and cautious orderings in this case. Recall from Example 13 that journey A was preferred to journey D, which was preferred to journey C, which was preferred to journey B. This means that to learn the preferences in Example 13, we could give the orderings $\langle e_A, e_D, < \rangle$, $\langle e_D, e_C, < \rangle$ and $\langle e_C, e_B, < \rangle$ as either brave or cautious orderings.

4.2 Context-dependent Learning from Answer Sets

Common to previous ILP frameworks is the underlying assumption that hypotheses should cover the examples with respect to one fixed given background knowledge. But, in practice, some examples may be context-dependent – different examples may need to be covered using different background knowledges. The journey preferences in Example 13 can be extended, for example, with contextual information (e.g. the weather).

Example 15. Reconsider the background knowledge and examples from Example 14. It may be that certain attributes of a journey are *context-dependent*; for instance, weather conditions may be important. Any of the ordering examples o in Example 14 could be extended with a context such as $C = \{\texttt{raining.}\}$. This would mean that for a brave ordering o, there should be a pair of answer sets of $B \cup H \cup C$ that extends the partial interpretations in o and that respects the ordering (w.r.t. the weak constraints in $B \cup H \cup C$).

In fact, the definition of a context-dependent ordering example given in this tutorial is slightly more general than in Example 15, as each partial interpretation in a context-dependent ordering example can have its own context. We will see that in addition to representing genuine contextual information, in some cases, contexts can be used in order to partition the background knowledge into pieces that are relevant to particular examples. We now formalise the notion of *context-dependent* examples. Similarly to ILP_{LOAS} examples, these are of two types: partial interpretations and ordering examples.

Definition 11. *A* context-dependent partial interpretation *(CDPI) is a pair $e = \langle e_{pi}, e_{ctx} \rangle$, where e_{pi} is a partial interpretation and e_{ctx} is an $\mathcal{ASP}^{ch}$ program (i.e. an ASP program with no weak constraints), called a* context. *Given a program P, an interpretation I is said to be an* accepting answer set *of e w.r.t. P if and only if $I \in AS(P \cup e_{ctx})$ and I extends e_{pi}. P is said to* accept *e if there is at least one accepting answer set of e w.r.t. P.*

Definition 12. *A* context-dependent ordering example *(CDOE) o is a tuple $\langle \langle e^1_{pi}, e^1_{ctx} \rangle, \langle e^2_{pi}, e^2_{ctx} \rangle, op \rangle$, where the first two elements are CDPIs and op is a binary comparison operator ($<$, $>$, $=$, $\leq$, $\geq$ or $\neq$). Given a CDOE $o = \langle e_1, e_2, op \rangle$, $inverse(o) = \langle e_1, e_2, op^{-1} \rangle$, where $<^{-1}$ is $\geq$, $\leq^{-1}$ is $>$, $=^{-1}$ is $\neq$, $\neq^{-1}$ is $=$, $>^{-1}$ is $\leq$ and $\geq^{-1}$ is $>$. A pair of interpretations $\langle I_1, I_2 \rangle$ is said to be an* accepting pair of answer sets *of o wrt a program P if all of the following conditions hold: (i) I_1 is an accepting answer set of $\langle e^1_{pi}, e^1_{ctx} \rangle$; (ii) I_2 is an accepting answer set of $\langle e^2_{pi}, e^2_{ctx} \rangle$; and (iii) $\langle I_1, I_2, op \rangle \in ord(P, AS(P \cup e^1_{ctx}) \cup AS(P \cup e^2_{ctx}))$. A program P is said to* bravely respect *o if there is at least one accepting pair of answer sets of o. P is said to* cautiously respect *o if there is no accepting pair of answer sets of $inverse(o)$.*

Definition 13. *A* Context-dependent Learning from Ordered Answer Sets *($ILP^{context}_{LOAS}$) task is a tuple $T = \langle B, S_M, \langle E^+, E^-, O^b, O^c \rangle \rangle$ where B is an ASP program, S_M is a set of ASP rules, E^+ and E^- are finite sets of CDPIs, and O^b and O^c are finite sets of CDOEs over E^+ called, respectively, brave and cautious orderings. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if:*

1. *$\forall e \in E^+$, $B \cup H$ accepts e*
2. *$\forall e \in E^-$, $B \cup H$ does not accept e*
3. *$\forall o \in O^b$, $B \cup H$ bravely respects o*
4. *$\forall o \in O^c$, $B \cup H$ cautiously respects o*

Example 16. Reconsider the journey preference learning task of Example 14. The contextual information in Example 15 can be added to the examples e_1 and e_2, to show the preference "in the case that it is raining e_1 is preferred to e_2, but otherwise it is the other way around" with the context dependent ordering examples o_1 and o_2:

$$o_1 = \left\langle \left\langle e_1, \left\{ \texttt{raining.} \right\} \right\rangle, \left\langle e_2, \left\{ \texttt{raining.} \right\} \right\rangle, < \right\rangle$$
$$o_2 = \left\langle \left\langle e_2, \emptyset \right\rangle, \left\langle e_1, \emptyset \right\rangle, < \right\rangle$$

4.3 ILASP

Inductive Learning of Answer Set Programs (ILASP) is a collection of algorithms for solving ILP_{LAS}, ILP_{LOAS} and $ILP^{context}_{LOAS}$ tasks. Similarly to ASPAL, each ILASP algorithm makes use of meta-level ASP programs. As we will see in Section 4.4, deciding whether a hypothesis is a solution of one of the ILP_b tasks solved by ASPAL is *NP*-complete in the propositional case, whereas the same decision problem for the tasks solved by ILASP is *DP*-complete. For this reason, ILASP 1 and 2 do not encode the search for solutions in a single meta-level ASP program (solving such a program is *NP*-complete in the propositional case), but instead employ an iterative algorithm, where a meta-level ASP program is solved repeatedly with new constraints added in each iteration, until the optimal answer sets of the meta-level program correspond to the optimal inductive solutions of the task.

The details of ILASP's meta-level programs are beyond the scope of this tutorial[6]. For the purposes of this tutorial, all the reader needs to know is that given any ILP_{LAS}, $ILP^{context}_{LAS}$ or $ILP^{context}_{LOAS}$ task T, both ILASP1(T) and ILASP2(T) return an optimal solution of T (resources permitting, of course).

Relevant examples and ILASP2i. The ILASP1 and ILASP2 algorithms both scale poorly with respect to the number of examples as the number of rules in the grounding of their meta-level ASP programs is proportional to the number of examples. The ILASP2i algorithm [24] solves a task iteratively, by building up a set of *relevant examples*. The idea is that in real tasks, many examples may be similar and may therefore be covered by exactly the same set of hypotheses. If this is the case, it is sufficient to consider only a small set of examples that are representative of the full set – these are the relevant examples. ILASP2i constructs this set iteratively, by assuming that its current relevant example set is completely representative of the full set, and using ILASP2 to solve the task with only those examples. If the assumption holds, then the hypothesis returned by ILASP2 will be an inductive solution of the full task. If not, then there must be at least one example which is not covered by the hypothesis returned by ILASP2 – this is added to the relevant example set before the next iteration.[7] The $findRelevantExample$ method is used to check whether a given hypothesis H is an inductive solution of the full task; if it is, then it returns `nil` (as there are no relevant examples to find); otherwise, it returns an example which is not covered by H.

In some ways, ILASP2i can be thought of as a non-monotonic variation on the idea of a cover-loop with three major differences: (1) just because an example is covered in one iteration, it is not guaranteed to be covered in future iterations

[6] Details of the encodings can be found in [21, 22, 20].

[7] In Algorithm 1.1 the set *Relevant* is a pair of sets of examples, the first set being relevant positive examples and the second set relevant negative examples. The notation on Line 5 means to add example re to the appropriate set, depending on whether it is a positive or a negative example.

Algorithm 1.1 ILASP2i

```
1: procedure ILASP2i(⟨B, S_M, E⁺, E⁻⟩)
2:     Relevant = ⟨∅, ∅⟩;   H = ∅;
3:     re = findRelevantExample(⟨B, S_M, E⁺, E⁻⟩, H);
4:     while re ≠ nil do
5:         Relevant << re;
6:         H = ILASP2(⟨B, S_M, Relevant⟩);
7:         if H == nil then
8:             return UNSATISFIABLE;
9:         else
10:            re = findRelevantExample(⟨B, S_M, E⁺, E⁻⟩, H);
11:        end if
12:    end while
13:    return H;
14: end procedure
```

(unless it is added to the set of relevant examples); (2) the learning starts from scratch in each iteration (rather than iteratively building a hypothesis); and (3) the full set of relevant examples is considered in each iteration (rather than a single current seed example).

4.4 The Complexity and Generality of Learning Answer Set Programs

Throughout this tutorial, we have discussed the six main frameworks for learning under the answer set semantics. As we introduced the early learning frameworks, we discussed some of their limitations, such as the fact that systems based on brave induction are unable to learn constraints. These limitations were some of the original motivations of the later frameworks such as ILP_{LAS}.

Although we have already demonstrated that there are programs which can be learned by ILP_{LAS} based systems that cannot be learned by systems based on earlier frameworks, it is more interesting to consider exactly which classes of programs can be learned by each framework. The aim is to characterise the class of ASP programs that a framework is capable of learning, if given sufficient examples. Language biases tend, in general, to impose their own restrictions on the classes of program that can be learned. They are primarily used to aid the performance of the computation, rather than to capture intrinsic properties of a learning framework. In this chapter we will therefore consider learning tasks with unrestricted hypothesis spaces: hypotheses can be constructed from any set of normal rules, choice rules and hard and weak constraints. We assume each learning framework $\mathcal{F}$ to have a task consisting of a pair $\langle B, E_{\mathcal{F}} \rangle$, where B is the (ASP) background knowledge and $E_{\mathcal{F}}$ is a tuple consisting of the examples for this framework; for example E_{LAS}[8] $= \langle E^+, E^- \rangle$ where E^+ and E^- are sets of partial interpretations.

[8] Note that to avoid cumbersome notation, we denote this E_{LAS} rather than $E_{ILP_{LAS}}$.

In [25], the generality of the six main frameworks was investigated and three new measures of generality were presented, based on which of the hypotheses a framework can *distinguish* from other hypotheses. Roughly speaking, a hypothesis H_1 can be distinguished from another hypothesis H_2 (with respect to a given background knowledge B) if there is at least one set of examples E such that $B \cup H_1$ satisfies every example in E and $B \cup H_2$ does not. The following definition formalises the *one-to-one-distinguishability* class of a learning framework.

Definition 14. *The* one-to-one-distinguishability class *of a learning framework $\mathcal{F}$ (denoted $\mathcal{D}_1^1(\mathcal{F})$) is the set of tuples $\langle B, H_1, H_2 \rangle$ of ASP programs for which there is at least one task $T_{\mathcal{F}} = \langle B, E_{\mathcal{F}} \rangle$ such that $H_1 \in ILP_{\mathcal{F}}(T_{\mathcal{F}})$ and $H_2 \notin ILP_{\mathcal{F}}(T_{\mathcal{F}})$. For each $\langle B, H_1, H_2 \rangle \in \mathcal{D}_1^1(\mathcal{F})$, $T_{\mathcal{F}}$ is said to* distinguish H_1 *from H_2 with respect to B.*

Note that the one-to-one-distinguishability relationship is not symmetric; i.e there are pairs of hypotheses H_1 and H_2 such that, given a background knowledge B, H_1 can be distinguished from H_2, but H_2 can not be distinguished from H_1. This is illustrated by Example 17.

Example 17. Consider a background knowledge B that defines the concepts of cell, same_block, same_row and same_column for a `4x4` Sudoku grid (see Example 11).

Let H_1 be the incomplete description of the Sudoku rules:

```
1 { value(C, 1), value(C, 2), value(C, 3), value(C, 4) } 1 :- cell(C).
:- value(C1, V), value(C2, V), same_row(C1, C2).
:- value(C1, V), value(C2, V), same_col(C1, C2).
```

Also let H_2 be the complete description of the Sudoku rules:

```
1 { value(C, 1), value(C, 2), value(C, 3), value(C, 4) } 1 :- cell(C).
:- value(C1, V), value(C2, V), same_row(C1, C2).
:- value(C1, V), value(C2, V), same_col(C1, C2).
:- value(C1, V), value(C2, V), same_block(C1, C2).
```

ILP_b can distinguish H_1 from H_2 with respect to B. This can be seen using the task $\langle B, \langle \{\texttt{value}((1,1),1), \texttt{value}((2,2),1)\}, \emptyset \rangle \rangle$. On the other hand, ILP_b cannot distinguish H_2 from H_1. Whatever examples are given in a learning task to learn H_2, it must be the case that $E^+ \subseteq A$ and $E^- \cap A = \emptyset$, where A is an answer set of $B \cup H_2$. But answer sets of $B \cup H_2$ are also answer sets of $B \cup H_1$. So A is also an answer set of $B \cup H_1$, which implies that H_1 satisfies the same examples and is a solution of the same learning task.

Table 1 gives conditions which are both sufficient and necessary for a tuple $\langle B, H_, H_1 \rangle$ to appear in the one-to-one-distinguishability class of each learning framework.[9] Proofs of the correctness of these conditions are given in [25]. The conditions show that that the following orderings hold:

[9] In Table 1 the following two notations are used. For programs P and Q the relation $P \equiv^s Q$ means that for any program R $AS(P \cup R) = AS(Q \cup R)$ and for a program P $\mathcal{E}_c(BP)$ is the set of conjunctions of literals in every answer set of P.

Framework $\mathcal{F}$	Sufficient/necessary condition for $\langle B, H_1, H_2\rangle$ to be in $\mathcal{D}^1_1(\mathcal{F})$
ILP_b	$AS(B \cup H_1) \not\subseteq AS(B \cup H_2)$
ILP_{sm}	$AS(B \cup H_1) \not\subseteq AS(B \cup H_2)$
ILP_c	$AS(B \cup H_1) \neq \emptyset \wedge (AS(B \cup H_2) = \emptyset \vee (\mathcal{E}_c(B \cup H_1) \not\subseteq \mathcal{E}_c(B \cup H_2)))$
ILP_{LAS}	$AS(B \cup H_1) \neq AS(B \cup H_2)$
ILP_{LOAS}	$(AS(B \cup H_1) \neq AS(B \cup H_2)) \vee (ord(B \cup H_1) \neq ord(B \cup H_2))$
$ILP^{context}_{LOAS}$	$(B \cup H_1 \not\equiv^s B \cup H_2) \vee (\exists C \in \mathcal{ASP}^{ch}$ s.t. $ord(B \cup H_1 \cup C) \neq ord(B \cup H_2 \cup C))$

Table 1. A summary of the sufficient and necessary conditions in each learning framework for a hypothesis H_1 to be distinguishable from another hypothesis H_2 with respect to a background knowledge B.

- $\mathcal{D}^1_1(ILP_b) = \mathcal{D}^1_1(ILP_{sm}) \subset \mathcal{D}^1_1(ILP_{LAS}) \subset \mathcal{D}^1_1(ILP_{LOAS}) \subset \mathcal{D}^1_1(ILP^{context}_{LOAS})$
- $\mathcal{D}^1_1(ILP_c) \subset \mathcal{D}^1_1(ILP_{LAS})$

If we view one-to-one-distinguishability as a measure of the generality of a learning framework, then ILP_b, ILP_{sm} and ILP_c are each strictly less general than ILP_{LAS}, and ILP_{LOAS} and $ILP^{context}_{LOAS}$ are more general still.

The one-to-many-distinguishability class of a learning framework. In practice, an ILP task has a search space of possible hypotheses, and it is important to know the cases in which one particular hypothesis can be distinguished from the rest. In what follows, we analyse the conditions under which a learning framework can distinguish a hypothesis from *a set* of other hypotheses. This corresponds to the notion of *one-to-many-distinguishability class* of a learning framework, which is a generalisation of the notion of the *one-to-one-distinguishability class*.

Definition 15. *The* one-to-many-distinguishability class *of a learning framework $\mathcal{F}$ (denoted $\mathcal{D}^1_m(\mathcal{F})$) is the set of all tuples $\langle B, H, \{H_1, \ldots, H_n\}\rangle$ such that there is a task $T_{\mathcal{F}}$ that distinguishes H from each H_i with respect to B.*

Given two frameworks $\mathcal{F}_1$ and $\mathcal{F}_2$, we say that $\mathcal{F}_1$ is at least as (resp. more) $\mathcal{D}^1_m$-general as (resp. than) $\mathcal{F}_2$ if $\mathcal{D}^1_m(\mathcal{F}_2) \subseteq \mathcal{D}^1_m(\mathcal{F}_1)$ (resp. $\mathcal{D}^1_m(F_2) \subset \mathcal{D}^1_m(F_1)$).

The one-to-many-distinguishability class tells us the circumstances in which a framework is general enough to distinguish some target hypothesis from a set of unwanted hypotheses. Note that, although the tuples in a one-to-many-distinguishability class that have a singleton set as the third argument correspond to the tuples in a one-to-one-distinguishability class of that framework, it is not always the case that if $\mathcal{F}_1$ is more $\mathcal{D}^1_m$-general than $\mathcal{F}_2$ then $\mathcal{F}_1$ is also more $\mathcal{D}^1_1$-general than $\mathcal{F}_2$. For example, we will see that ILP_{sm} is more $\mathcal{D}^1_m$-general than ILP_b, but we have already seen that the ILP_b and ILP_{sm} are equally $\mathcal{D}^1_1$-general.

Example 18. $\mathcal{D}^1_m(ILP_b) \subset \mathcal{D}^1_m(ILP_{sm})$. We can see this as follows. Firstly, clearly $\mathcal{D}^1_m(ILP_b) \subseteq \mathcal{D}^1_m(ILP_{sm})$, as any ILP_b task can be trivially mapped into an ILP_{sm} task. Thus, it remains to show that $\mathcal{D}^1_m(ILP_b) \neq \mathcal{D}^1_m(ILP_{sm})$.

Consider the programs $B = \emptyset$, $H = \{\texttt{1\{heads, tails\}1.}\}$, $H_1 = \{\texttt{heads.}\}$ and $H_2 = \{\texttt{tails.}\}$. $\langle B, H, \{H_1, H_2\}\rangle \in \mathcal{D}^1_m(ILP_{sm})$ ($\langle B, \langle\{\langle\{\texttt{tails}\}, \emptyset\rangle, \langle\{\texttt{heads}\}, \emptyset\rangle\}\rangle\rangle$ distinguishes H from H_1 wrt the background knowledge B). We now show that there is no task $T_b = \langle B, \langle E^+, E^-\rangle\rangle$ such that $H \in ILP_b(T_b)$ and $\{H_1, H_2\} \cap ILP_b(T_b) = \emptyset$.

Assume for contradiction that there is such a task T_b. As $H \in ILP_b(T_b)$ and $AS(B \cup H) = \{\{\texttt{heads}\}, \{\texttt{tails}\}\}$, $E^+ \subset \{\texttt{heads}, \texttt{tails}\}$ and $E^- \subset \{\texttt{heads}, \texttt{tails}\}$ (neither can be equal to $\{\texttt{heads}, \texttt{tails}\}$ or H would not be a solution).

Case 1: $E^+ = \emptyset$

Case a: $E^- = \emptyset$

Then H_1 and H_2 would be inductive solutions. This is a contradiction as $\{H_1, H_2\} \cap ILP_b(T_b) = \emptyset$.

Case b: $E^- = \{\texttt{heads}\}$

Then H_2 would be an inductive solution of T_b. Contradiction.

Case c: $E^- = \{\texttt{tails}\}$

Then H_1 would be an inductive solution of T_b. Contradiction.

Case 2: $E^+ = \{\texttt{heads}\}$

$\texttt{heads} \notin E^-$ as otherwise the task would have no solutions (and we know that H is a solution). In this case H_1 would be an inductive solution (regardless of what else is in E^-). Contradiction.

Case 3: $E^+ = \{\texttt{tails}\}$

Similarly to above case, $\texttt{tails} \notin E^-$ as otherwise the task would have no solutions. In this case H_2 would be an inductive solution (regardless of what else is in E^-). Contradiction.

Hence, there is no such task $T_b = \langle B, \langle E^+, E^-\rangle\rangle$ such that $H \in ILP_b(T_b)$ and $\{H_1, H_2\} \cap ILP_b(T_b) = \emptyset$. So, $\mathcal{D}^1_m(ILP_b) \neq \mathcal{D}^1_m(ILP_{sm})$.

In [25], it is shown that the following orderings hold.

- $\mathcal{D}^1_m(ILP_b) \subset \mathcal{D}^1_m(ILP_{sm}) \subset \mathcal{D}^1_m(ILP_{LAS}) \subset \mathcal{D}^1_m(ILP_{LOAS}) \subset \mathcal{D}^1_m(ILP^{context}_{LOAS})$
- $\mathcal{D}^1_m(ILP_c) \subset \mathcal{D}^1_m(ILP_{LAS})$

[25] presents a further measure of generality, many-to-many-distinguishability. The many-to-many-distinguishability class of a framework is used to analyse which sets of hypotheses can be distinguished from other sets of hypotheses. However, the many-to-many-distinguishability class is outside the scope of this tutorial.

Complexity. Given the differences in generality between the various learning frameworks, an obvious question to ask is whether there is any price to pay in terms of computational complexity when using the more general frameworks. In this section, we consider three common decision problems when using the learning frameworks:

- *Verification*: deciding whether a given hypothesis is an inductive solution of a given learning task.

Framework	Verification	Satisfiablity	Optimum Verification
ILP_b	NP-complete	NP-complete	DP-complete
ILP_{sm}	NP-complete	NP-complete	DP-complete
ILP_c	DP-complete	Σ_2^P-complete	Π_2^P-complete
ILP_{LAS}	DP-complete	Σ_2^P-complete	Π_2^P-complete
ILP_{LOAS}	DP-complete	Σ_2^P-complete	Π_2^P-complete
$ILP_{LOAS}^{context}$	DP-complete	Σ_2^P-complete	Π_2^P-complete

Table 2. A summary of the complexity of the various learning frameworks. *Verification* corresponds to deciding whether a given hypothesis is a solution of a given learning task. *Satisfiability* corresponds to deciding whether a learning task has any solutions at all. *Optimum verification* corresponds to deciding whether a given hypothesis is the optimal (shortest) solution of a given task.

- *Satisfiability*: deciding whether a given learning task has any inductive solutions.
- *Optimum Verification*: deciding whether a given hypothesis is an optimal inductive solution of a given learning task.

Table 2 gives the complexity results for propositional versions of each of the learning frameworks (where the background knowledge, contexts of examples and hypothesis space is restricted to propositional ASP). Proofs of the results in Table 2 can be found in [20]. Interestingly despite the great difference in the generality of the various frameworks, for each of the three decision problems, $ILP_{LOAS}^{context}$ has the same complexity as ILP_c. The complexity of both ILP_b and ILP_{sm} is lower than any of the other frameworks, which suggests that in applications where the increased generality of the other frameworks is not needed, ILP_{sm} may be more suitable. It should be noted that ILASP may still be used to solve such tasks – ILP_{LAS} tasks with no negative examples are equivalent to ILP_{sm} tasks.

4.5 Learning Answer Set Programs from Noisy Examples

The learning from answer sets frameworks have recently been upgraded to support learning from noisy examples [26]. In this section, we present a generalisation of the idea to give a general way of upgrading any non-noisy learning framework with a notion of penalised examples. There are already several algorithms, predating these formal definitions, which adopt the approach of penalising examples (e.g. XHAIL [36] and Inspire [18]).

Given any learning framework ILP_F covered in this tutorial (ILP_b, ILP_c, ILP_{LAS}, etc) a task of the penalised framework $n(ILP_F)$ is of the same form as tasks for ILP_F, other than the fact that each example is of the form $e@p$, where e is an example of the previous framework and p is either ∞ (meaning the example must be covered) or it is a positive integer representing the *penalty* for not covering that example. This penalty is also often called a *weight* for the example.

Given any task T and hypothesis H, the score of H w.r.t. T, written $\mathcal{S}(H,T)$, is equal to $|H| + \sum_{e@p \in U} p$ where U is the set of examples in T that are not covered by H. In the case of brave induction (where each answer set of $B \cup H$ might suggest that different examples are covered), $\mathcal{S}(H,T)$ is assigned the minimum possible score. In the case of cautious induction, for the penalty of a hypothesis to be finite, it must also be satisifiable when it is combined with the background knowledge. The inductive solutions of a task are the hypotheses with a finite score. The optimal inductive solutions are the set of inductive solutions which minimise the score.

Example 19. Consider an extension of the ILP_b task from Example 3, $T' = \langle B, S_M, \langle E^+, E^- \rangle\rangle$, where:

$$B = \left\{\begin{array}{l}\texttt{bird(X) :- penguin(X).}\\ \texttt{bird(X) :- sparrow(X).}\\ \texttt{penguin(b1).}\\ \texttt{penguin(b2).}\\ \texttt{penguin(b3).}\\ \texttt{sparrow(b4).}\end{array}\right\} \qquad \begin{array}{l} E^+ = \{\texttt{flies(b1)@2}, \texttt{flies(b4)@2}\} \\ E^- = \{\texttt{flies(b2)@2}, \texttt{flies(b3)@2}\} \\ S_M = \left\{\begin{array}{l}\texttt{h}_1 : \texttt{flies(X) :- bird(X).}\\ \texttt{h}_2 : \texttt{flies(X) :- bird(X),}\\ \qquad\qquad \texttt{not penguin(X).}\end{array}\right\}\end{array}$$

- $\mathcal{S}(\emptyset, T') = |\emptyset| + 4 = 4$.
- $\mathcal{S}(\{\texttt{h}_1\}, T') = |\{\texttt{h}_1\}| + 4 = 5$ (recall that the *type* atom `bird(X)` does not count towards the length of the rule).
- $\mathcal{S}(\{\texttt{h}_2\}, T') = |\{\texttt{h}_2\}| + 2 = 4$.
- $\mathcal{S}(\{\texttt{h}_2\}, T') = |\{\texttt{h}_1, \texttt{h}_2\}| + 4 = 7$.

This task has two optimal inductive solutions: $\emptyset$ and $\{\texttt{h}_2\}$. The choice of penalty for the examples is important. If each of the examples in this task had had penalty 1, $\emptyset$ would have been optimal; whereas if the penalties had all been 3, $\{\texttt{h}_2\}$ would have been optimal.

Note that we have used an extremely small hypothesis space here to keep things simple. In reality, the hypothesis space would usually be much bigger!

The ASPAL encoding shown in the previous section can be extended to solve noisy tasks. This is achieved using weak constraints to represent the penalties of the examples. The XHAIL and ILASP systems have also been extended to handle noise in a similar way by using optimisation in ASP. ASPAL and ILASP are both guaranteed to find an optimal inductive solution of any task; however, as shown in Example 20 XHAIL may not.

Example 20. Consider the following noisy task, in the XHAIL input format:

```
p(X) :- q(X, 1), q(X, 2).          #modeh r(+s).
p(X) :- r(X).                      #modeh q(+s2, +t).
s(a).   s(b).   s2(b).             #example not p(a)=50.
t(1).   t(2).                      #example p(b)=100.
```

This corresponds to a hypothesis space that contains two facts $F_1 = \texttt{r(X)}$, $F_2 = \texttt{q(X,Y)}$ (in XHAIL, these facts are implicitly "typed", so the first fact, for example, can be thought of as the rule $\texttt{r(X):-s(X)}$). The two examples have penalties 50 and 100 respectively. There are four possible hypotheses: $\emptyset$, F_1, F_2 and $F_1 \cup F_2$, with scores 100, 51, 1 and 52 respectively. XHAIL terminates and returns F_1, which is a suboptimal hypothesis.

The issue is with the first step. The system finds the smallest abductive solution, $\{\texttt{r(b)}\}$ and as there are no body declarations in the task, the kernel set contains only one rule: $\texttt{r(b):-s(b)}$. XHAIL then attempts to generalise to a first order hypothesis that covers the examples. There are two hypotheses which are subsets of a generalisation of $\texttt{r(b)}$ (F_1 and $\emptyset$); as F_1 has a lower score than $\emptyset$, XHAIL terminates and returns F_1. The system does not find the abductive solution $\{\texttt{q(b,1)}, \texttt{q(b,2)}\}$, which is larger than $\{\texttt{r(b)}\}$ and is therefore not chosen, even though it would eventually lead to a better solution than $\{\texttt{r(b)}\}$.

It should be noted that XHAIL does have an *iterative deepening* feature for exploring non-minimal abductive solutions, but in this case using this option XHAIL still returns F_1, even though F_2 is a more optimal hypothesis. Even when iterative deepening is enabled, XHAIL only considers non-minimal abductive solutions if the minimal abductive solutions do not lead to any non-empty inductive solutions.

Although ILASP1, ILASP2 and ILASP2i are all guaranteed to find optimal inductive solutions of any $n(ILP_{LOAS}^{context})$ task, they do not perform well when solving tasks with noise. ILASP3 is specifically targetted at learning tasks with noisy examples; however, a discussion of ILASP3 is beyond the scope of this tutorial. An in depth discussion of ILASP3 can be found in [20], and an evaluation of ILASP3 on several noisy datasets can be found in [26].

5 Conclusion

This tutorial has presented an introduction to logic based learning under the answer set semantics. We have introduced the six main frameworks for learning ASP programs, and presented generality results highlighting the flaws in early frameworks and showing that to learn some ASP programs the recent, more general, frameworks are required. The development of learning frameworks has been matched by the development of more sophisticated algorithms, of which we have given an overview in this tutorial. The most recent ILASP system supports learning ASP programs including normal rules, choice rules and hard and weak constraints, even from noisy examples. However, there are still challenges to be addressed, particularly with respect to scalability, which is the focus of our current research.

References

1. Athakravi, D.: Inductive logic programming using bounded hypothesis space. Ph.D. thesis, Imperial College London (2015)

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. Blockeel, H., De Raedt, L.: Top-down induction of first-order logical decision trees. Artificial intelligence **101**(1), 285–297 (1998)
4. Brain, M., Cliffe, O., De Vos, M.: A pragmatic programmers guide to answer set programming. Answer Set Programming p. 49 (2009)
5. Corapi, D., Russo, A.: ASPAL. proof of soundness and completeness. Tech. rep., Department of Computing (DTR11-5), Imperial College, London (2011)
6. Corapi, D., Russo, A., Lupu, E.: Inductive logic programming as abductive search. In: ICLP (Technical Communications). pp. 54–63 (2010)
7. Corapi, D., Russo, A., Lupu, E.: Inductive logic programming in answer set programming. In: Inductive Logic Programming, pp. 91–97. Springer (2012)
8. Dastani, M., Jacobs, N., Jonker, C.M., Treur, J.: Modeling user preferences and mediating agents in electronic commerce. In: Agent Mediated Electronic Commerce, pp. 163–193. Springer (2001)
9. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: A primer. In: Reasoning Web. Semantic Technologies for Information Systems, pp. 40–110. Springer (2009)
10. Erdem, E., Gelfond, M., Leone, N.: Applications of answer set programming. AI Magazine **37**(3), 53–68 (2016)
11. Fürnkranz, J., Hüllermeier, E.: Pairwise preference learning and ranking. In: Machine Learning: ECML 2003, pp. 145–156. Springer (2003)
12. Geisler, B., Ha, V., Haddawy, P.: Modeling user preferences via theory refinement. In: Proceedings of the 6th international conference on Intelligent user interfaces. pp. 87–90. ACM (2001)
13. Gelfond, M., Kahl, Y.: Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach. Cambridge University Press (2014)
14. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. vol. 88, pp. 1070–1080 (1988)
15. Horváth, T.: A model of user preference learning for content-based recommender systems. Computing and informatics **28**(4), 453–481 (2012)
16. Inoue, K., Kudoh, Y.: Learning extended logic programs. In: IJCAI (1). pp. 176–181 (1997)
17. Katzouris, N., Artikis, A., Paliouras, G.: Incremental learning of event definitions with inductive logic programming. Machine Learning **100**(2-3), 555–585 (2015)
18. Kazmi, M., Schüller, P., Saygın, Y.: Improving scalability of inductive logic programming via pruning and best-effort optimisation. Expert Systems with Applications **87**, 291–303 (2017)
19. Kowalski, R., Sergot, M.: A logic-based calculus of events. New generation computing **4**(1), 67–95 (1986)
20. Law, M.: Inductive Learning of Answer Set Programs. Ph.D. thesis, Imperial College London (2018)
21. 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)
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.: Simplified reduct for choice rules in ASP. Tech. rep., Department of Computing (DTR2015-2), Imperial College London (2015)

24. 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)
25. Law, M., Russo, A., Broda, K.: The complexity and generality of learning answer set programs. Artificial Intelligence **259**, 110–146 (2018)
26. Law, M., Russo, A., Broda, K.: Inductive learning of answer set programs from noisy examples. Advances in Cognitive Systems (2018)
27. Mueller, E.T.: Commonsense reasoning: An event calculus based approach. Morgan Kaufmann (2014)
28. Muggleton, S.: Inductive logic programming. New generation computing **8**(4), 295–318 (1991)
29. Muggleton, S.: Inverse entailment and progol. New generation computing **13**(3-4), 245–286 (1995)
30. Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K., Srinivasan, A.: ILP turns 20. Machine Learning **86**(1), 3–23 (2012)
31. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog decision support system for the space shuttle. In: Practical Aspects of Declarative Languages, pp. 169–183. Springer (2001)
32. Nuffelen, B.: Abductive constraint logic programming: implementation and applications. Ph.D. thesis, K.U.Leuven (2004)
33. Otero, R.P.: Induction of stable models. In: Inductive Logic Programming, pp. 193–205. Springer (2001)
34. Papadimitriou, C.H.: Computational complexity. John Wiley and Sons Ltd. (2003)
35. Ray, O.: Hybrid abductive inductive learning. Ph.D. thesis, Imperial College London (2005)
36. Ray, O.: Nonmonotonic abductive inductive learning. Journal of Applied Logic **7**(3), 329–340 (2009)
37. Ray, O., Broda, K., Russo, A.: Hybrid abductive inductive learning: A generalisation of progol. In: Inductive Logic Programming, pp. 311–328. Springer (2003)
38. Ricca, F., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A logic-based system for e-tourism. Fundamenta Informaticae **105**(1-2), 35–55 (2010)
39. Sakama, C.: Inverse entailment in nonmonotonic logic programs. In: ILP. pp. 209–224. Springer (2000)
40. Sakama, C.: Nonmonotomic inductive logic programming. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 62–80. Springer (2001)
41. Sakama, C., Inoue, K.: Brave induction: a logical framework for learning from incomplete information. Machine Learning **76**(1), 3–35 (2009)
42. Seitzer, J., Buckley, J.P., Pan, Y.: Inded: A distributed knowledge-based learning system. IEEE Intelligent Systems and their Applications **15**(5), 38–46 (2000)
43. Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In: International Symposium on Practical Aspects of Declarative Languages. pp. 305–319. Springer (1999)
44. Srinivasan, A.: The Aleph manual. Machine Learning at the Computing Laboratory, Oxford University (2001)