Learning from Answer Sets via Single-Shot Disjunctive ASP Encoding

Roberto Borelli[1, 2], **Agostino Dovier**[1]

[1]University of Udine, Dept of Mathematics, Computer Science, and Physics
[2]University of Padua, Dept of Mathematics
roberto.borelli@phd.unipd.it, agostino.dovier@uniud.it

Abstract

Deep Learning techniques are nowadays pervasive in AI. However, these approaches suffer from a lack of transparency for justifying their output and for helping users in believing in their decisions. For these reasons alternative approaches to learning deserve to be explored either for developing new tools with autonomous learning capability or for explaining the results of black-box predictors. Among them an important role is assumed since the Nineties by Inductive Logic Programming and, in particular, recently by the approaches of Learning from Answer Sets (LAS). Computing inductive solutions for LAS tasks is known to be Σ_2^P-hard. In this work, we tackle this problem using a single-shot disjunctive ASP encoding based on the saturation technique originally proposed by Eiter and Gottlob. We prove that, when the background knowledge and hypothesis space form a *tight* program (a syntactical property) our encoding is linear in the size of the task. This approach contrasts with the state-of-the-art ILASP system, which relies on multiple iterative calls to an ASP solver. As a result, it can be directly evaluated by modern disjunctive ASP solvers, leveraging decades of research and optimization in the ASP community. We implement our method in a system named LASCO. Experimental results on a diverse set of benchmarks demonstrate that LASCO outperforms all versions of ILASP on many instances and it scales if run on multi-threaded machines.

LASCO Code & Appendix — https://clp.dimi.uniud.it/sw/

1 Introduction

The ability to learn interpretable models from data is a long-standing goal of Artificial Intelligence. While sub-symbolic methods such as Deep Learning have achieved remarkable empirical performance, they often lack transparency and explanatory power, making them inadequate for safety-critical applications that demand trustworthy and explainable decisions. In this regard, *Inductive Logic Programming (ILP)* (Muggleton 1991) offers an alternative, symbolic approach to machine learning, where hypotheses are learned in the form of logic programs. Formally, an *ILP task* T is defined as a tuple $\langle B, S, E\rangle$, where B is a logic program called the *background knowledge*, S is a set of rules called the *hypothesis space*, and E is a set of examples. An *inductive solution* $H \subseteq S$ is a hypothesis such that the program $B \cup H$ "entails" all examples in E. A significant advance in ILP was introduced by Law (2018), who proposed the *Learning from Answer Sets* (LAS) framework. The background knowledge is represented by an ASP program (without disjunctive heads), and the learning task involves two distinct sets of examples: the set of positive examples E^+ and the set of negative examples E^-. Recent research has shown that combining ILP (particularly the LAS framework) with neural networks can enhance explainability in hybrid neuro-symbolic architectures (Cunnington et al. 2023; Dreossi 2024; Dreossi et al. 2024, 2025; Dozzi et al. 2025). Learning in the LAS framework poses significant computational challenges. Specifically, computing inductive solutions of a given LAS task is proved to be a Σ_2^P-hard problem. Existing LAS solvers tackle this problem by using an iterative strategy: optimal inductive solutions are found with multiple invocations of an ASP solver.

In this work, we present a novel approach to solve LAS tasks via a single-shot disjunctive ASP encoding, which incorporates the entire learning task into an ASP program. Our encoding is based on the saturation technique introduced by Eiter and Gottlob (1995) and is linear under the assumption that the union of background knowledge and hypothesis space forms a tight program, i.e., one with no loops in the positive atom dependency graph. This single-shot design enables the ASP solver to fully leverage its internal optimization techniques - such as propagation, intelligent grounding, and conflict-driven clause learning, by having a complete global view of the problem. By offloading the reasoning process to a mature ASP solver, our approach benefits from decades of research, theoretical analysis, and performance engineering embedded in systems like CLINGO (Gebser et al. 2022) and DLV (Alviano et al. 2010). In contrast, ILASP with its own custom optimization layers, effective in many cases, cannot yet match the robustness, generality, and efficiency of state-of-the-art ASP engines.

To deal with non-ground tasks, we introduce a notion of single-shot grounding for LAS tasks thus permitting the encoding to work with the class of safe tasks exactly as the ILASP system does. Building on these foundations, we implemented a new LAS solver named LASCO that supports configurable solving backends, encoding strategies, and search modalities, while maintaining compatibility with

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

the ILASP input format. We evaluated LASCO on a variety of relevant benchmarks demonstrating competitive performance, in general, and superior performance on many instances. The paper is structured as follows: after providing some background in Sect. 2 and related work in Sect. 3, in Sect. 4, we introduce the single-shot encoding, proving its soundness and completeness. In Sect. 5, we deal with grounding issues and in Sect. 6, we present the LASCO system that is empirically compared against ILASP in Sect. 7. Conclusions are discussed in Sect. 8.

2 Background

We recall ASP syntax, semantics, the computational complexity of standard reasoning tasks, and introducing the notation adopted in this work. Finally, we introduce the Learning from Answer Sets framework.

Answer Set Programming (ASP) A *signature* is a tuple $\Sigma = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$, where $\mathcal{P}$ is a set of predicate symbols, $\mathcal{V}$ a set of variables, and $\mathcal{C}$ a set of constant symbols. A *term* is either a constant or a variable. An *atomic formula* (or simply, an *atom*) is an expression of the form $p(t_1, \ldots, t_{ar(p)})$, where $p \in \mathcal{P}$ is a predicate symbol, each t_i is a term, and $ar(p)$ denotes the number of arguments of p. An atom is said to be *ground* if it contains no variables. A *literal* is either an atom a, or $\texttt{not}\ \ a$, where "not" denotes *negation as failure (naf)*. An *ASP program* P over Σ consists of a finite set of *general disjunctive rules* of the form:

$$\overbrace{H_1 \ \mid \ \ldots \ \mid \ H_k}^{head(R)} \ \texttt{:-} \ \underbrace{A_1, \ldots, A_n, \texttt{not}\ \ B_1, \ldots, \texttt{not}\ \ B_m}_{body(R)}.$$

where each H_i, A_j, and B_k are atoms, and $k, n, m \geq 0$. Logically, a rule expresses an implication: *if the body holds, then the head must also hold.* The *head* of the rule, denoted *head*(R), is the disjunction of atoms $H_1, \ldots, H_k$. The *body*, denoted *body*(R), is the conjunction of literals $A_1, \ldots, A_n$ and $\texttt{not}\ \ B_1, \ldots, \texttt{not}\ \ B_m$. We denote with *body*$^+(R)$ (resp., *body*$^-(R)$) the set of positive (resp., negative) atoms in the body of R. A rule is said to be *normal* if its head contains exactly one atom ($k = 1$). *atoms*$(P) := \bigcup_{R \in P} head(R) \cup body^+(R) \cup body^-(R)$ denotes the set of all atoms in P. A program is said to be *ground* if it contains only ground atoms. A program P is *safe* if for every rule $R \in P$, every variable X appearing in R, also appears in some literal in *body*$^+(R)$. We denote by ASP^D the class of ASP programs consisting of general disjunctive rules and by ASP^N its subclass of consisting only of normal rules.

The semantics of ASP is designed to model commonsense reasoning, which is inherently non-monotonic. This means that adding new rules to a program P can invalidate previously derived conclusions. The *Herbrand universe* of an ASP program P, denoted with HU_P, is the set of ground terms generated from a signature Σ, in this case, simply the set of constant symbols. The *Herbrand base* of an ASP program P, denoted with HB_P, is the set of all ground atoms that can be formed by applying the predicate symbols in $\mathcal{P}$ to all possible ground terms in HU_P. An *Herbrand interpretation* I is a subset of the Herbrand base HB_P, i.e., a truth assignment over atoms. A rule R if satisfied by I if either at least one of the $A_i \notin I$ or at least one of the $B_j \in I$ or at least one of the $H_k \in I$. If all rules are satisfied by I then $I \models P$ (we say that I is a *model* of P). An interpretation I is a *minimal Herbrand model* of P if no proper subset $J \subsetneq I$ is also a model P. We look for models of P that are called *answer sets* or *stable models*, a concept introduced by Gelfond and Lifschitz (1988). Given an interpretation I, the reduct of P w.r.t. I, denoted P^I, is obtained from P by removing all rules whose body is not satisfied by the atoms in I, and removing all naf-literals from the remaining rules. An interpretation I is a *stable model* (or answer set) of a ground ASP program P if and only if I is a minimal model of P^I. The set of stable models of P is denoted by *AS*(P). Given a program P a literal ℓ is *bravely* entailed if there is a stable model of P in which ℓ holds. A literal ℓ is *cautiously* entailed if it holds in all stable models of P, and there is at least one of them.

To compute the answer sets of a non-ground program P we first compute its *grounding*, denoted *ground*(P), obtained by replacing the variables in P with all possible combinations of ground terms from HU_P. I is an answer set of P if and only if I is an answer set of *ground*(P).

Given an ASP program P, the *consistency problem*, denoted *AS*$(P) \neq \emptyset$, asks to decide whether or not P admits a stable model. The following well-known theorem establishes the complexity of this problem.

Theorem 2.1 *Let P be a ground ASP program.*

- *If $P \in \mathsf{ASP}^\mathsf{N}$, deciding AS$(P) \neq \emptyset$ is* NP-complete*;*
- *If $P \in \mathsf{ASP}^\mathsf{D}$, deciding AS$(P) \neq \emptyset$ is* Σ_2^P-complete.

The result on ASP^D programs is due to a famous proof by Eiter and Gottlob (1995) which introduces the *saturation* technique, which will be applied in our encoding (Sect. 4).

We now show that answer sets of ASP^N programs can be characterized by means of Boolean formulas following the approach presented by Lin and Zhao (2004).

Let P be a ground ASP^N program. $\phi_{\mathit{iff}}(P)$, is defined as:[1]

$$\phi_{\mathit{iff}}(P) := \bigwedge_{a \in atoms(P)} (a \leftrightarrow \bigvee_{R \in P,\, head(R) = a} BF(R))$$

where *BF*(R), the body formula w.r.t. rule R, is defined as

$$BF(R) := \bigwedge_{b \in body^+(R)} b \wedge \bigwedge_{b \in body^-(R)} \neg b$$

Let P be an ASP program. The positive atom dependency graph $G(P)$ is defined as $G(P) := (atoms(P), E)$ where $E = \{(b, h) \mid R \in P, b \in body^+(R), h \in head(R)\}$. Let P be an ASP program, and let E be the set of edges of $G(P)$. A set of atoms L is a loop of P if it induces a non-trivial strongly connected subgraph of $G(P)$, namely, each pair of atoms in L is connected by a non-empty path in the graph $(L, E \cap (L \times L))$. We denote with *loops*(P) the set of loops

[1] Clark's Equality Theory (Clark 1978) should be considered here; since we are using a signature without function symbols, with a slight abuse of notation, we call $\phi_{\mathit{iff}}(P)$ the completion of P.

of P. We say that P is *tight* if $loops(P) = \emptyset$. Let P be a ground ASP^N program. The loop formula of P, is:

$$\phi_{loop}(P) \coloneqq \bigwedge_{L \in loops(P)} (\bigvee_{a \in L} a \to \bigvee_{R \in ES(L)} BF(R))$$

where $ES(L)$ is a set of rules called *external support* (with respect to the loop L) and is defined as:

$$ES(L) \coloneqq \{R \in P \mid head(R) \in L, body^+(R) \cap L = \emptyset\}$$

The following theorem from (Lin and Zhao 2004), relates answer sets of a ground ASP^N program P to the models of a Boolean formula $\phi_{as}(P)$.

Theorem 2.2 *Let P be a ground ASP^N program and let $I \subseteq atoms(P)$. It holds that $I \in AS(P)$ if and only if I is a model of $\phi_{as}(P) \coloneqq \phi_{iff}(P) \wedge \phi_{loop}(P)$.*

Learning from Answer Sets *Learning from Answer Sets* (LAS) is a ILP framework introduced by Law (2018). In LAS, the background knowledge is represented by an ASP^N program, and the hypothesis space consists of ASP^N rules. Each example is labeled as either positive or negative, and these two types of examples are treated asymmetrically: a hypothesis must *bravely* entail all positive examples and *cautiously* not entail all negative ones. This asymmetric treatment allows LAS to generalize previous frameworks based on brave and cautious induction (e.g. (Sakama and Inoue 2009)).

Examples are represented using the following concepts. A *partial interpretation* e is a pair of sets of ground atoms $\langle e^{incl}, e^{excl}\rangle$. An interpretation I *extends* e if and only if $e^{incl} \subseteq I$ and $e^{excl} \cap I = \emptyset$.

We can now formalize the central notion of an ILP_{LAS} task. A *learning from answer sets* (ILP_{LAS}) task T is a tuple $\langle B, S, E^+, E^-\rangle$ where E^+ and E^- are sets of partial interpretations. A hypothesis $H \subseteq S$ is an inductive solution of T if and only if

1. $\forall e \in E^+ \; \exists A \in AS(B \cup H) \; A$ extends e, and
2. $\forall e \in E^- \; \forall A \in AS(B \cup H) \; A$ does not extend e.

Let X be an ILP framework and T be an ILP_X task. We denote by $ILP_X(T)$ the set of inductive solutions of T, and by ${}^*ILP_X(T)$ the set of optimal inductive solutions of T, where optimality is defined in terms of minimal hypothesis length (i.e., $|H|$). Consider now the case of T being an ILP_{LAS} task. We say that T is *ground* (resp. *safe*) if $B \cup S$ is ground (resp. safe). The *satisfiability problem* asks whether T admits at least one inductive solution.

Theorem 2.3 (Mark Law 2018) *Deciding satisfiability for ground ILP_{LAS} tasks is Σ_2^P-complete.*

The Σ_2^P-hardness comes from the presence of negative examples: it can be proved that the complexity of the satisfiability problem for LAS tasks containing only positive examples (to be bravely entailed) drops to NP-complete.

3 Related Work

We don't survey here the history of Inductive Logic programming that traces back to (Muggleton 1991). Our focus is instead on ILASP, Inductive Learning of Answer Set Programs (Law, Russo, and Broda 2020), together with other algorithms proposed for variants of the LAS framework. ILASP1 incrementally searches for optimal hypotheses by increasing their length. At each iteration i, it performs two ASP solver calls: first, to find all violating hypotheses (those covering negative examples); second, to exclude them via constraints and compute valid hypotheses of size i. If any valid hypothesis remains, it is returned as an optimal solution; otherwise, the process continues with $i + 1$. The *brave encoding*, later discussed in section 4, is inspired by the encoding used in the ILASP1 algorithm. ILASP2 introduces violating reasons to optimize the search, while ILASP2i improves scalability by iteratively focusing only on relevant examples. ILASP3 handles noisy examples through approximate coverage, and ILASP4 (Law 2023) formalizes Conflict-Driven ILP using an iterative procedure involving hypothesis search, counterexample search, conflict analysis, and constraint propagation.

FastLAS and FastLAS2 (Law et al. 2020; 2021) are two algorithms aimed at achieving scalable learning from answer sets. The final stage of both algorithms consists of solving a single-shot encoding. A key difference with our single-shot encoding, presented in section 4 and implemented in the LASCO system, is that FastLAS supports only a restricted fragment of the ILP_{LAS} framework. Although FastLAS is not directly comparable to systems such as ILASP and LASCO (which support the full framework) it is noteworthy that FastLAS allows scoring functions to handle noisy data and achieves good scalability by exploring only a restricted portion of the hypothesis space defined via mode declarations, rather than enumerating it entirely.

Popper (Cropper and Morel 2020) is another popular ILP system based on the *learning from failures* methodology. One of its main advantages is that it avoids a global grounding step, which enables better scalability as the domain size increases. Nevertheless, Popper is restricted to learning definite programs, i.e., programs without negation.

4 Encoding

We present a single-shot ASP^D encoding capable of dealing with a ground ILP_{LAS} task $T = \langle B, S, E^+, E^-\rangle$; we define the ASP^D program $P_{dis}(T)$ such that the set $AS(P_{dis}(T))$ corresponds one-to-one with the set of inductive solutions of T. If the program $B \cup S$ is tight, the encoding is linear in the size of T; otherwise, the size of the encoding may become exponential due to the inclusion of loop formulas. The program $P_{dis}(T)$ is composed of two main components: (i) $P^+_{dis}(T)$, which ensures that all positive examples are bravely entailed; (ii) $P^-_{dis}(T)$, which enforces that all negative examples are cautiously avoided. First we describe these two components in detail, then we discuss soundness, completeness and complexity of the encoding.

Cautious encoding Given a ground ILP_{LAS} task T, the cautious entailment component $P^-_{dis}(T)$ is a ground ASP^D program constructed in two main stages:

1. The task T is first translated into a quantified Boolean

formula $\phi(T)$ of the form $\exists X \forall Y\ \psi(T)$.

2. $\phi(T)$ is then compiled into the ASP^D program $P^-_{dis}(T)$.

The first stage builds upon Theorem 2.2, while the second stage relies on the proof of the second point of Theorem 2.1.

Stage 1 We start by defining the "guarded" hypothesis space S': for a given rule $h_i \in S$, we define its counterpart $h'_i \in S'$ by associating it a (new) ground atom `h(i)`. Precisely, if h_i is $H \leftarrow B$, then h'_i is $H \leftarrow B,$ `h(i)`.

The idea is that a rule in S' can be triggered solely when its guard `h(i)` is activated. Let $P(T) := B \cup S$, $P'(T) := B \cup S'$, and $guards := \{\texttt{h(1)}, \texttt{h(2)}, \dots\}$. The objective is to represent the answer sets of $P'(T)$ as a Boolean expression after fixing the truth values for all atoms `h(i)`.

Although the formula $\phi_{as}(P'(T))$, discussed in Theorem 2.2, may seem a plausible choice, this is not the case. Since the atoms `h(i)` are absent in the head of any rule, they would be all set to false (according to the stable model semantics). Instead, we want these atoms to remain externally unrestricted, as an oracle could independently assign their values. Thus, we outline revised formulas $\phi^*_{iff}(\cdot)$, $\phi^*_{loop}(\cdot)$, and $\phi^*_{as}(\cdot)$, as follows:

- $\phi^*_{iff}(P)$ differs from $\phi_{iff}(P)$ only in the scope of the first conjunction: $a \in atoms(P) \setminus guards$
- Similarly $\phi^*_{loop}(P)$ differs from $\phi_{loop}(P)$ in the scope of the first disjunction: $a \in L \setminus guards$

The formula $\phi^*_{as}(P)$ is then defined as the conjunction of $\phi^*_{iff}(P)$ and $\phi^*_{loop}(P)$. As a consequence of Theorem 2.2, we establish the correspondence between models of $\phi^*_{as}(P'(T))$ and the answer sets of $P'(T)$ extended with fixed guards.

Lemma 4.1 *Let A be an interpretation of $\phi^*_{as}(P'(T))$, then:*

$$A \models \phi^*_{as}(P'(T)) \leftrightarrow A \in AS(P'(T) \cup facts(guards \cap A))$$

where, for a set J, $facts(J)$ is the ASP program $\bigcup_{j \in J}\{\texttt{j.}\}$.

Let $V_{P'(T)}$ denote the set of Boolean variables appearing in the Boolean formula $\phi^*_{as}(P'(T))$ (i.e., $V_{P'(T)}$ has one variable for each atom in the Herbrand base of $P'(T)$). We further partition $V_{P'(T)} = V_{P(T)} \cup V^H_{P'(T)}$, where:
(i) $V_{P(T)}$ is the set of variables of $\phi_{as}(P(T))$, and
(ii) $V^H_{P'(T)} = guards \cap S = \{\texttt{h(i)} \mid h_i \in S\}$ represents the set of guard atoms for the hypotheses.

We now introduce a propositional formula that characterizes the entailment of the negative example set. Recall that each negative example $e \in E^-$ is a partial interpretation of the form $\langle e^{incl}, e^{excl}\rangle$. We define the formula $\phi_{neg}(E^-)$ as:

$$\phi_{neg}(E^-) := \bigwedge_{e \in E^-} \Big(\bigvee_{a \in e^{incl}} \neg a \vee \bigvee_{a \in e^{excl}} a\Big).$$

This formula ensures that, for each negative example, at least one inclusion atom is absent or one exclusion atom is present - i.e., an interpretation satisfying $\phi_{neg}(E^-)$ fails to extend every example in E^-. We now consider the formula $\psi(T)$ defined as:

$$\psi(T) := \mathsf{NNF}(\phi^*_{as}(P'(T)) \rightarrow \phi_{neg}(E^-))$$

where $\mathsf{NNF}(F)$ denotes the *Negation Normal Form* (NNF) of the formula F, that is, a logically equivalent formula in which negation is applied only to atomic propositions, and the only connectives allowed are $\wedge$, $\vee$, and $\neg$. Notice that the set of free variables of $\psi(T)$ is $V_{P'(T)}$. Let $\{y_1, \dots, y_v\}$ be the set of variables $V_{P(T)}$ and let s denote the number of rules in S. We define the formula $\phi(T)$ as follows:

$$\phi(T) := \exists h(1) \dots \exists h(s) \forall y_1 \dots \forall y_v\ \psi(T).$$
$$= \exists V^H_{P'(T)}\ \forall V_{P(T)}\ \psi(T)$$

The idea behind $\phi(T)$ is the following: once a hypothesis $H \subseteq S$ is fixed (i.e., we assign truth values to the set of existential variables $V^H_{P'(T)} = \{h(1), \dots, h(s)\}$), we then consider all possible truth assignments to the universal variables $V_{P(T)} = \{y_1, \dots, y_v\}$. Whenever the complete assignment to the variables $V_{P'(T)} = V^H_{P'(T)} \cup V_{P(T)}$ satisfies the formula $\phi^*_{as}(P'(T))$, it corresponds to an answer set $A \in AS(B \cup H)$. In this case, we require that A does not extend any negative example, that is, the formula $\phi_{neg}(E^-)$ must also be satisfied.

Stage 2 From the quantified Boolean formula $\phi(T)$, we construct the program $P^-_{dis}(T)$ by applying a translation similar to the one used by Eiter and Gottlob in the proof of Theorem 2.1. The resulting ASP^D program encodes the formula $\phi(T) = \exists V^H_{P'(T)}\ \forall V_{P(T)}\ \psi(T)$ via saturation. In particular the program $P^-_{dis}(T)$ is defined as follows.

Definition 4.2 (Cautious encoding $P^-_{dis}(T)$)

`h(i) | nh(i).` $\quad \forall h(i) \in V^H_{P'(T)}$
`y | ny. y :- w. ny :- w.` $\quad \forall y \in V_{P(T)}$
`w :- formula`$_{\psi(T)}$`.`
expansion$(\psi(T))$
`w :- not w.`

The auxiliary predicate `formula`$_{\psi(T)}$ is used to evaluate the formula $\psi(T)$ in a bottom-up fashion over its syntax tree. The rule `w :- not w` enforces that `w` must be true in every stable model; however, due to the stable model semantics, `w` must also be supported by `formula`$_{\psi(T)}$. The subroutine *expansion*(F) encodes the semantics of the Boolean formula F (assumed to be in NNF) as ASP rules, using auxiliary atoms of the form `formula`$_F$:

if F is a variable $p \in V_{P'(T)}$**:** *expansion*(F) is

`formula`$_F$ `:- p.`

if F is $\neg p$ for a variable $p \in V_{P'(T)}$**:** *expansion*(F) is

`formula`$_F$ `:- np.`

if F is $F_1 \wedge F_2$: *expansion*(F) is

`formula`$_F$ `:- formula`$_{F_1}$`, formula`$_{F_2}$`.`
expansion(F_1) $\quad$ *expansion*(F_2)

if F is $F_1 \vee F_2$: *expansion*(F) is

`formula`$_F$ `:- formula`$_{F_1}$`.`
`formula`$_F$ `:- formula`$_{F_2}$`.`
expansion(F_1) $\quad$ *expansion*(F_2)

Brave encoding Given a ground ILP_{LAS} task T, the brave entailment component $P^{+}_{dis}(T)$ uses elements similar to the encoding used in the ILASP1 algorithm.

Definition 4.3 (Brave encoding $P^{+}_{dis}(T)$) *For each rule $R_i \in B$ of the form H `:-` $A_1, \ldots, A_n$, `not` $B_1, \ldots,$ `not` B_m. the program $P^{+}_{dis}(T)$ contains the rule Bg(i):*

```
inAs(H,M) :- inAs(A_1,M),...,inAs(A_n,M),
    not inAs(B_1,M),...,not inAs(B_m,M),
    interpretation(M).
```

For each hypothesis $h_i \in S$ of the form H `:-` $A_1, \ldots, A_n$, `not` $B_1, \ldots,$ `not` B_m. the program $P^{+}_{dis}(T)$ contains the rule Hypo(i) defined as:

```
inAs(H,M) :- inAs(A_1,M),...,inAs(A_n,M),
    not inAs(B_1,M),...,not inAs(B_m,M),
    interpretation(M),h(i).
```

Finally, for each positive example $e_i \in E^+$ of the form $\langle \{e^{incl}_1, \ldots, e^{incl}_n\}, \{e^{excl}_1, \ldots, e^{excl}_m\}\rangle$, the program $P^{+}_{dis}(T)$ includes the rules $Ex_1(i)$, $Ex_2(i)$ and $Ex_3(i)$:

```
interpretation(i).
cov(i) :- inAs(e_1^incl,i),...,inAs(e_n^incl,i)
   not inAs(e_1^excl,i),...,not inAs(e_m^excl,i).
:- not cov(i).
```

Intuitively, for each positive example, we are looking for an answer set of $B \cup H$ that satisfies it. To this end, for every positive example e_i, we introduce a fresh atom `interpretation(`i`)`, which identifies a candidate answer set for that example. We then encode the rules from the background knowledge and hypothesis space using atoms of the form `inAs(`A, M`)`, where M is a variable ranging over the possible answer set identifiers. This allows us to simulate the construction of an answer set for each example separately. In the case of hypothesis rules, we also guard them with the atom `h(i)`, which is true only if the corresponding rule $h_i \in S$ is selected.

Final encoding The final encoding $P_{dis}(T)$ is defined as the union of the two modules:

$$P_{dis}(T) \coloneqq P^{-}_{dis}(T) \cup P^{+}_{dis}(T).$$

This disjunctive encoding follows a guess-and-verify approach. The module $P^{-}_{dis}(T)$ guesses a hypothesis $H \subseteq S$ that satisfies the constraints imposed by the negative examples. Then, the module $P^{+}_{dis}(T)$ uses the guessed hypothesis and checks whether it also satisfies all the positive examples. In practice, the set of inductive solutions for T is given by:

$$\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}(T))\}.$$

Alternatively, this set can be obtained by intersecting the inductive solutions computed by $P^{-}_{dis}(T)$ with those computed by $P^{+}_{dis}(T)$. The next theorem formalizes correctness and completeness of the single-shot encoding.

Theorem 4.4 (Correctness and completeness) *Let $T = \langle B, S, E^+, E^-\rangle$ be a ground ILP_{LAS} task. It holds that $ILP_{LAS}(T) = \{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}(T))\}$.*

To conclude our discussion of the single-shot encoding, the next lemma analyzes the structural complexity of the program $P_{dis}(T)$.

Lemma 4.5 *Let $T = \langle B, S, E^+, E^-\rangle$ be a ground ILP_{LAS} task. Then the following holds:*

1. $|P_{dis}(T)| \in \Theta\left(|T| + |\phi^{*}_{loop}(P'(T))|\right)$
2. $|ground(P_{dis}(T))| \in \Theta(\,|B \cup S| \times |E^+| + |E^-| + |B| + |S| + |\phi^{*}_{loop}(P'(T))|\,)$
3. *If $B \cup S$ is tight then* $|P_{dis}(T)| \in \Theta(|T|)$ *and* $|ground(P_{dis}(T))| \in \Theta(|B \cup S| \times |E^+| + |E^-| + |B| + |S|)$.

5 Grounding

So far, our encoding $P_{dis}(T)$ has been designed for ground ILP_{LAS} tasks. However, ground tasks are often too restrictive for real-world applications. The main difficulty with non-ground tasks is that, given a task $T = \langle B, S, E^+, E^-\rangle$, the grounding of $B \cup H$ may differ for each subset $H \subseteq S$. In this section we show that for the class of *safe* non-ground ILP_{LAS} tasks, the grounding of the full program $B \cup S$ is sufficient to compute inductive solutions. The main problem of the proposed approach is that we require the full instantiation of the program $B \cup S$. Thus, we cannot take advantage of advanced grounding techniques available in tools like GRINGO and DLV.

We begin by introducing the *aggregate LAS* framework, denoted as ILP^{agg}_{LAS}. This framework generalizes ILP_{LAS} by allowing each hypothesis element $h_i \in S$ to be a *set* of $\mathsf{ASP}^{\mathsf{N}}$ rules (an aggregation of rules) rather than a single $\mathsf{ASP}^{\mathsf{N}}$ rule, as in the standard setting. Formally, for an ILP^{agg}_{LAS} task T, a hypothesis $H \subseteq S$ is an inductive solution of T if and only if

1. $\forall e \in E^+ \; \exists A \in AS(B \cup \bigcup_{h_i \in H} h_i)$ A extends e;
2. $\forall e \in E^- \; \forall A \in AS(B \cup \bigcup_{h_i \in H} h_i)$ A does not extend e.

Let X and Y be two ILP frameworks and let T_1 (resp. T_2) be an ILP_X (resp. ILP_Y) task with hypothesis space S_1 (resp. S_2). We say that T_1 and T_2 are *equivalent* if there exists a bijection $f : 2^{S_1} \to 2^{S_2}$ such that $H_1 \subseteq S_1$ is an inductive solution of T_1 if and only if $f(H_1)$ is an inductive solution of T_2.

The aggregate LAS framework is clearly a generalization of the standard LAS setting. Given a ground ILP_{LAS} task $T = \langle B, S, E^+, E^-\rangle$, we can verify that $T^{agg} := \langle B, \{\{h_i\} \mid h_i \in S\}, E^+, E^-\rangle$ is equivalent to T.

Our goal is to transform a non-ground ILP_{LAS} task T and into a ground ILP^{agg}_{LAS} task *ground*(T) such T and *ground*(T) are equivalent.

Definition 5.1 (Grounding for ILP_{LAS} tasks) *Let $T = \langle B, S, E^+, E^-\rangle$ be a (possibly non-ground) ILP_{LAS} task. Let P be the program $B \cup S$. Let U be the Herbrand universe of P. We define ground(T), the grounded version of T as the following ILP^{agg}_{LAS} task:*

$$\langle \bigcup_{R_i \in B} ground_U(R_i), \{ground_U(h_i) \mid h_i \in S\}, E^+, E^-\rangle$$

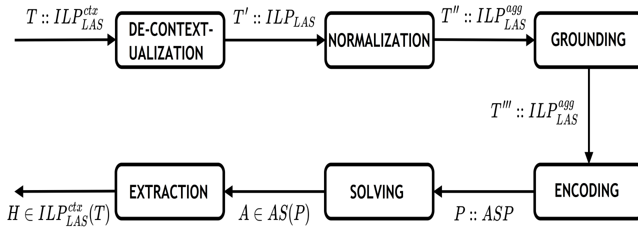

Figure 1: Main steps performed by LASCO.

where $ground_U(R)$ *denotes the set of ground instances of* R *using all possible terms in* U.

The main theorem that we prove is that every safe ILP_{LAS} task T is equivalent to its grounded version.

Theorem 5.2 *Let* T *be a (possibly non-ground) safe* ILP_{LAS} *task. Let* $T' = ground(T)$. *It holds that* T *and* T' *are equivalent.*

Since our encoding $P_{dis}(T)$ can be adapted to handle ground ILP^{agg}_{LAS} tasks, we have that the composition $P_{dis}(ground(T))$ is a correct and complete encoding for T, provided that T is a safe ILP_{LAS} task.

6 LASCO Pipeline

We present now the tool LASCO, designed to solve LAS tasks via the single-shot encoding technique. The workflow of LASCO consists of six main phases, as illustrated in Figure 1. The input to the system is a task T expressed in the ILP^{ctx}_{LAS} framework, which is a slight extension of the standard ILP_{LAS} framework in which each example is augmented with an associated *context*.

1. **De-contextualization.** The input task T is transformed into an equivalent task T', formulated in the standard ILP_{LAS} framework. This transformation was originally introduced by (Law 2018) and eliminates context information by embedding it into the background knowledge.
2. **Normalization.** The task T' is then converted into an equivalent task T'' in the ILP^{agg}_{LAS} framework, where every rule is a normal rule. T'' belongs to the ILP^{agg}_{LAS} framework rather than the standard ILP_{LAS} one because certain ASP constructs (such as choice rules) are rewritten as sets of normal rules (see e.g. (Gebser et al. 2022)).
3. **Grounding.** In this step, the task T'' is grounded, yielding a fully instantiated task T''' in the ILP^{agg}_{LAS} framework (c.f. Section 5). The grounding procedure is implemented via a call to GRINGO (Gebser, Schaub, and Thiele 2007), that produces the complete grounding of the input program (definition 5.1).
4. **Encoding.** The grounded task T''' is then compiled into an ASP program P, such that there exists a one-to-one correspondence between the answer sets of P and the inductive solutions of T''', and thus of the original task T. This translation is performed using the single-shot disjunctive encoding $P_{dis}(\cdot)$ introduced in section 4.
5. **Solving.** This phase computes the answer sets of the ASP program P, using an external ASP solver such as CLINGO (Gebser et al. 2014) or DLV (Alviano et al. 2010). If one seeks optimal inductive solutions of T, optimization statements can be included in P, and the solver can be instructed to return only optimal answer sets. Moreover, the number of solutions to be computed can be specified depending on the desired number of inductive hypotheses.
6. **Extraction.** Finally, given an answer set A of P, an inductive hypothesis H_A solving the original task T is computed as $H_A = \{h_i \in S \mid \texttt{h(i)} \in A\}$.

7 Evaluation

In this section, we compare[2] the performance of LASCO with the state-of-the-art LAS system ILASP4 and with ILASP2i, an earlier version specifically engineered to scale well with respect to the number of examples. We point out two differences between LASCO and ILASP: (1) LASCO indirectly supports multi-threading, as the generated ASP encoding can be evaluated using CLINGO with multiple threads; (2) LASCO can be configured to return non-optimal inductive solutions, enabling trade-offs between runtime and optimality. By contrast, ILASP supports only single-threaded executions and always returns optimal inductive solutions, which may lead to significant overhead in large or complex tasks. In the tables, we report execution times in *seconds*, truncated at a timeout of 3600 seconds (indicated by *e*). In the tests LASCO is configured to find optimal inductive solutions using 1, 2, 4, 8, and 16 threads (indicated by 1t,...,16t). We also include a variant of LASCO configured with 16 threads to return the first (possibly non-optimal) inductive solution (indicated by 16tf).

Table 1 presents our first benchmark, which consists of ten learning tasks designed to *infer* the binary relations $<$, $=$, and $>$ over integer numbers. Each task includes exactly one positive example, fewer than 100 negative examples, and a hypothesis space containing fewer than 150 rules. Instances are annotated using the following convention: *T* indicates that the hypothesis space is *tight* (i.e., contains no loops); *L* indicates the presence of loops, with fewer than 10000 in total; *S* denotes satisfiable instances, while *U* indicates unsatisfiable ones. Moreover, the name of each instance includes an integer number k, indicating that the task considers only the first k positive integers. LASCO outperforms both ILASP2i and ILASP4 on every instance, even when running in single-threaded mode. Let us also observe that ILASP4 is consistently faster than ILASP2i across all instances. In addition, LASCO exhibits excellent scalability with respect to the number of threads. For example, instances UT4 and ST4 exceed the 1-hour timeout when executed with a single thread, but are solved in under 100 seconds using 16 threads. This improvement is a direct consequence of using CLINGO as a parallel backend solver. When configured to return the first inductive solution (instead of the optimal one), LASCO achieves substantial speedups.

[2]All tests were run on an Ubuntu 24.04 desktop machine with a 13th Gen Intel(R) Core(TM) i7-13700KF (24 threads) processor and with 128 GB RAM. We used CLINGO version 5.6.2.

	ILASP2i	ILASP4	LASCO1t	LASCO2t	LASCO4t	LASCO8t	LASCO16t	LASCO16tf
ST2	0.33	0.18	0.05	0.06	0.06	0.07	0.08	0.08
ST3	54.88	23.24	2.32	1.20	0.53	0.48	0.35	0.33
ST4	e	e	e	1679.80	203.49	178.94	92.28	18.80
UT2	0.36	0.19	0.05	0.06	0.05	0.08	0.06	0.06
UT3	68.14	24.25	3.23	0.79	0.54	0.49	0.36	0.35
UT4	e	e	e	1211.33	211.27	203.30	85.15	101.91
SL2	0.44	0.23	0.09	0.08	0.08	0.08	0.09	0.08
SL3	422.01	347.84	223.18	96.84	65.48	54.71	45.32	21.12
UL2	0.51	0.27	0.10	0.09	0.08	0.08	0.10	0.09
UL3	e	339.84	216.18	73.28	77.26	61.30	72.83	43.96

Table 1: Runtimes for integer learning tasks.

Table 2 presents our second benchmark, which includes 15 tasks introduced in (Ielo et al. 2023). Each task is designed to learn a Linear Temporal Logic (LTL) formula over finite traces, from a set of positive and negative traces encoded as examples. In the original work, the authors show that ILASP2i is capable of outperforming SAT-based approaches on the LTL passive learning problem. Each of the 15 tasks contains approximately one thousand positive examples and no negative ones. In this context LASCO uses only the brave encoding part. The results in Table 2 highlight several key findings. First, ILASP4 is unable to solve any instance within the one-hour timeout, while LASCO, even when restricted to a single thread, solves 13 out of 15 instances within that limit. Second, ILASP2i consistently outperforms LASCO across all instances in this benchmark. Although LASCO shows some improvement when using 16 threads, the speedup is less pronounced than in table 1, where multi-threading yielded performance improvements of up to an order of magnitude. Finally, while configuring LASCO to compute the first (non-optimal) solution reduces the runtime in some cases, occasionally halving it, this strategy does not lead to substantial improvements across the entire benchmark. The main reason why ILASP2i outperforms LASCO on instances with a large number of examples is that ILASP2i is an *active learner*: it incrementally selects and processes only a subset of all the examples. In contrast, LASCO adopts a *batch learning* approach, where all examples are considered simultaneously from the beginning of the computation.

Overall, both benchmarks demonstrate that LASCO can be substantially faster than ILASP4, which is architecturally the most similar system to LASCO due to its use of conflict analysis. However, while ILASP4 incorporates conflict analysis internally, LASCO relies entirely on the underlying ASP solver (e.g., CLINGO) for hypothesis space exploration. This design enables LASCO to benefit from native conflict-driven clause learning (CDCL) and propagation strategies implemented in modern ASP solvers. Despite the conceptual similarities, LASCO consistently outperforms ILASP4 across all tested benchmarks in terms of execution time. This empirical evidence supports the idea that generating a single-shot, disjunctive ASP encoding upfront, and delegating both search and optimization to a highly engineered solver, is not only a viable strategy, but a promising and effective approach for solving ILP_{LAS} tasks efficiently.

	ILASP2i	ILASP4	LASCO1t	LASCO16t	LASCO16tf
AF	1.01	e	1228.76	1198.53	1192.94
NE	0.71	e	2006.41	2001.83	1990.67
NA	0.73	e	2144.89	2061.96	2027.09
IMSI-1	1.10	e	1656.02	1687.91	1679.57
IMSI-3	6.14	e	2221.93	1962.50	1839.38
IM	0.84	e	2008.28	2039.80	2036.20
AKA	48.71	e	1607.31	758.36	411.28
BT	0.32	e	88.52	85.10	65.05
EMM	1.10	e	1897.54	1980.48	1932.87
CWP	4.15	e	347.73	249.75	127.50
GLBA	18.41	e	538.31	349.88	125.10
HIPPA-1	10.33	e	492.98	299.61	158.40
HIPPA-2	0.50	e	110.50	121.74	96.78
MR	226.66	e	e	e	2220.77
RLF	258.45	e	e	e	e

Table 2: Runtimes for passive LTL learning tasks.

8 Conclusion and Future Work

In this work, we addressed the Σ_2^P problem of computing solutions for Learning from Answer Sets (LAS) tasks. Existing systems, such as ILASP, tackle this challenge by relying on multiple encodings and iterative invocations of an ASP solver, often employing domain-specific optimizations at each stage. In contrast, we proposed a single-shot strategy: the entire LAS task is compiled into a single disjunctive ASP program, from which inductive solutions (or optimal ones) are extracted directly from its answer sets. We first focused on ground LAS tasks, introducing an encoding based on disjunctive ASP and leveraging the classical saturation technique of Eiter and Gottlob (1995). We then extended our approach to non-ground tasks.

The main contribution is the implementation of the LASCO solver, which we evaluated against the state-of-the-art ILASP system. Our experimental results demonstrate that LASCO generally outperforms ILASP4, especially when taking advantage of multi-thread parallelism. We plan to explore also how GPU parallelism for ASP (see, e.g., Dovier et. al 2015; 2016; 2018) can be exploited. There are instances in which ILASP2i achieves better performance. This behavior is primarily due to its active learning architecture, which processes only a subset of relevant examples at each iteration. Several directions remain open for future research. To improve scalability with respect to the number of examples, we plan to investigate how incremental solving, available in modern ASP solvers such as CLINGO, can be integrated into our single-shot framework to simulate an active learning strategy similar to that of ILASP2i. We intend to extend our encodings with mode declarations, as done in systems like FastLAS, to better manage large and complex hypothesis spaces. The more general approach ASP(Q) (Faber et al. 2023) can be used for Σ_2^P problems; its applicability on LAS deserves to be investigated. Finally, we leave the reader with an open theoretical question: *Can every ground ILP_{LAS} task be compiled into a polynomial-size disjunctive ASP program?* We conjecture that the answer is negative, based on the fact that there is no polynomial-size Boolean encoding for arbitrary ground ASP programs.

Acknowledgments

The authors wish to thank Talissa Dreossi, Andrea Formisano, Enrico Pontelli, and Enrico Santi for the many interesting discussions on the material presented in this paper. The research was partially supported by Interdepartment Project on AI (Strategic Plan of UniUD–22-25), and by Unione europea-Next Generation EU, missione 4 componente 2, project MaPSART "Future Artificial Intelligence (FAIR)".

References

Alviano, M.; Faber, W.; Leone, N.; Perri, S.; Pfeifer, G.; and Terracina, G. 2010. The Disjunctive Datalog System DLV. In de Moor, O.; Gottlob, G.; Furche, T.; and Sellers, A. J., eds., *Datalog Reloaded - First International Workshop, Datalog, Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, 282–301. Springer.

Clark, K. L. 1978. Negation as Failure. In Gallaire, H.; and Minker, J., eds., *Logic and Databases*, 293–321. Plenum Press.

Cropper, A.; and Morel, R. 2020. Learning programs by learning from failures. arXiv:2005.02259.

Cunnington, D.; Law, M.; Lobo, J.; and Russo, A. 2023. Neuro-symbolic learning of answer set programs from raw data. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, IJCAI '23. ISBN 978-1-956792-03-4.

Dal Palù, A.; Dovier, A.; Formisano, A.; and Pontelli, E. 2015. CUD@SAT: SAT solving on GPUs. *Journal of Experimental and Theoretical Artificial Intelligence*, 27(3): 293 – 316.

Dovier, A.; Formisano, A.; and Pontelli, E. 2018. Parallel answer set programming. In *Handbook of Parallel Constraint Reasoning*, 237–282. Springer.

Dovier, A.; Formisano, A.; Pontelli, E.; and Vella, F. 2016. A GPU implementation of the ASP computation. *Lecture Notes in Computer Science*, 9585: 30–47.

Dozzi, M.; Dreossi, T.; Baron, L.; Costantini, F.; Dovier, A.; and Formisano, A. 2025. Semi-automatic knowledge representation and reasoning on vague crime concepts. In *Computational Intelligence and Mathematics for Tackling Complex Problems 6*. Springer.

Dreossi, T. 2024. Bridging Logic Programming and Deep Learning for Explainability through ILASP. In Cabalar, P.; Fabiano, F.; Gebser, M.; Gupta, G.; and Swift, T., eds., *Proceedings 40th International Conference on Logic Programming, ICLP*, volume 416 of *EPTCS*, 314–323.

Dreossi, T.; Dovier, A.; Formisano, A.; Law, M.; Manzato, A.; Russo, A.; and Tait, M. 2024. Towards Explainable Weather Forecasting Through FastLAS. In Dodaro, C.; Gupta, G.; and Martinez, M. V., eds., *Logic Programming and Nonmonotonic Reasoning - 17th International Conference, LPNMR, Proceedings*, volume 15245 of *Lecture Notes in Computer Science*, 262–275. Springer.

Dreossi, T.; Dovier, A.; Urli, S.; Pause, F. C.; and Crociati, M. 2025. Explainable AI for Sperm Morphology: Integrating YOLO with FastLAS. In Guidotti, D.; Pandolfo, L.; and Pulina, L., eds., *Proceedings of the 40th Italian Conference on Computational Logic*, volume 4003 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Eiter, T.; and Gottlob, G. 1995. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Ann. Math. Artif. Intell.*, 15(3-4): 289–323.

Faber, W.; Mazzotta, G.; and Ricca, F. 2023. An Efficient Solver for ASP(Q). *Theory and Practice of Logic Programming*, 23(4): 948–964.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR*, abs/1405.3694.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2022. *Answer set solving in practice*. Springer Nature.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. GrinGo : A New Grounder for Answer Set Programming. In Baral, C.; Brewka, G.; and Schlipf, J. S., eds., *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, 266–271. Springer.

Gelfond, M.; and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In Kowalski, R.; Bowen; and Kenneth, eds., *Proceedings of International Logic Programming Conference and Symposium*, 1070–1080. MIT Press.

Ielo, A.; Law, M.; Fionda, V.; Ricca, F.; De Giacomo, G.; and Russo, A. 2023. Towards ILP-Based LTL Passive Learning. In Bellodi, E.; Lisi, F. A.; and Zese, R., eds., *Inductive Logic Programming*, 30–45. Cham: Springer Nature Switzerland. ISBN 978-3-031-49299-0.

Law, M. 2018. *Inductive learning of answer set programs*. Ph.D. thesis, Imperial College London, UK.

Law, M. 2023. Conflict-Driven Inductive Logic Programming. *Theory and Practice of Logic Programming*, 23(2): 387–414.

Law, M.; Russo, A.; Bertino, E.; Broda, K.; and Lobo, J. 2020. Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, 2877–2885.

Law, M.; Russo, A.; and Broda, K. 2020. The ilasp system for inductive learning of answer set programs. *arXiv preprint arXiv:2005.00904*.

Law, M.; Russo, A.; Broda, K.; and Bertino, E. 2021. Scalable Non-observational Predicate Learning in ASP. In *IJCAI*, 1936–1943.

Lin, F.; and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2): 115–137.

Muggleton, S. H. 1991. Inductive Logic Programming. *New Gener. Comput.*, 8(4): 295–318.

Sakama, C.; and Inoue, K. 2009. Brave induction: a logical framework for learning from incomplete information. *Machine Learning*, 76: 3–35.