Exploring ILASP Through Logic Puzzles Modelling*

Talissa Dreossi[1]

[1]University of Udine, DMIF, Via delle Scienze 206, 33100 Udine, Italy

Abstract

ILASP (Inductive Learning of Answer Set Programs) is a logic-based machine learning system. It makes use of existing knowledge base, containing anything known before the learning starts or even previously learned rules, to infer new rules. We propose a survey on how ILASP works and how it can be used to learn constraints. In order to do so we modelled different puzzles in Answer Set Programming: the main focus concerns how different datasets can influence the learning algorithm and, consequently, what can or cannot be learnt.

Keywords

ILASP, logic programming, learning constraints, ASP

1. Introduction

In the last few decades, Machine Learning is arousing more and more interest. The main techniques to be mentioned are Artificial Neural Networks, Reinforcement Learning and Recommendation Systems. Note that all these techniques are commonly used as black box: they give a solution, with a certain accuracy, but cannot explain how they got it. Machine Learning has been also analysed within Logic Programming [1]. The main advantage of Logic-based Machine Learning is in fact its explainability [2]: known and learned knowledge are expressed as a set of logical rules that can be translated into natural language. The importance of the AI branch involved, the Explainable Artificial Intelligence one, is increasing in parallel with AI advancement. The reason is that in order to use AI algorithm in several situations of real life we need to give a justification of the prediction we receive from the system.

In this paper we will explore the functionality of ILASP, an Inductive Logic Program framework [3] developed as part of Mark Law's PhD thesis [4]. The following section gives a theoretical background on Inductive Logic Programming and then explains how ILASP is able to 'learn' logic programs. In Section 4, some logic puzzles are presented with their corresponding ASP program written by ourselves, as was made for example in [5], [6] and [7]. Continuing to Section 5 we explain the dataset, i.e. the examples, we used to get new constraints. The paper closes showing the results we got and some general considerations.

CILC'23: 38th Italian Conference on Computational Logic, June 21–23, 2023, Udine, Italy

*T. Dreossi is member of the INdAM Research group GNCS. Research partially supported by *Interdepartment Project on AI* (Strategic Plan of UniUD–2022-25)

dreossi.talissa@spes.uniud.it (T. Dreossi)

2. Related Works

Several approaches have been proposed for inductive learning of logic programs. We discuss some of the well-known methods that have contributed to the field, providing context for understanding ILASP. In this section, we will also explore some of the key applications of ILASP and its contribution to solving real-world problems.

In this paper we show how ILASP is capable of learning puzzles' rules. In fact, ILP was often exploit to learn rules for games such as chess [8], Bridge [9] and sudoku [10].
Nevertheless, ILASP can been used also in other fields including automated planning and control systems, in order to learn control policies or planning strategies. These applications has been adopted, for example, in robotics for surgical tasks [11]. Logic programming is a suitable option for task planning in robot-assisted surgery because it enables trusted reasoning using domain knowledge and improves decision-making transparency. Application concerning explainability was also made in [12] by Fossemò et al.. They usesd a dataset of users preferences over a set of recipes to train different neural networks (NNs). The aim was to conduct preliminary explore how ILASP can globally approximate these NNs. Since computational time required for training ILASP on high dimensional feature spaces is very high, they decided to focus also on how it would be possible to make global approximation more scalable. Their solution involved the use of Principal Component Analysis to reduce the dataset's dimensionality while maintaining transparent explanations. A similar approach was employed by D'Asaro et al. [13], where ILASP was used to explain the output of SVM classifiers trained on preference datasets. The distinction from the previous study lies in the production of explanations in terms of weak constraints as well.
Another area of interest is generative policies. Generative policies have been proposed as a mechanism to learn the constraints and preferences of a system within a specific context, allowing the system to coherently adapt to unexpected changes achieving the system goals with minimal human intervention. In [14], Calo et al. present the main workflow for global and local policy refinement and their adaptation based on Answer Set Programming (ASP) for policy representation and reasoning using ILASP.

3. ILASP

ILASP (Inductive Learning of Answer Set Programs) [4][10] is a logic-based machine learning system. It make use of existing knowledge base, containing anything known before the learning starts or even previously learned rules, to infer new rules. Unlike many other ILP systems, which are only able to learn definite logic programs, ILASP is able to learn normal rules, choice rules and hard and weak constraints (that is the subset of ASP accepted by ILASP). We use the term *rule* to refer to any of these four components. This means that it can be applied to preference learning, learning common-sense knowledge, including defaults and exceptions, and learning non-deterministic theories.

We give now a brief overview of Inductive Logic Programming (ILP) [15] and then we show a framework of ILP that learns from answer sets.

3.1. Inductive Logic Programming

ILP can use different theoretical settings as learning framework: *Learning From Entailment* (LFE), *Learning From Interpretations* (LFI), and *Learning From Satisfiability* (LFS). A particular problem associated with a framework is called a *learning task* [4] and will be denoted by T. A learning task T will consist of a background knowledge B, which is a logic program, a hypothesis space S, defining the set of rules allowed to be in a hypothesis, and some examples, whose form depends on the learning frameworks. In this section, a formal definition is given for each one.

Definition 3.1 (LFE)**.** A *Learning from Entailment* (ILP$_{LFE}$) task T is a tuple $\langle B, S, E^+, E^- \rangle$ where B is a *clausal theory*, called the background knowledge, S is a set of clauses, called the *hypothesis space* and E^+ and E^- are sets of ground clauses, called the *positive* and *negative examples*, respectively. A *hypothesis* $H \subseteq S$ is an *inductive solution* of T if and only if:

- $B \cup H \models E^+$,
- $B \cup H \cup E^- \not\models \bot$.

where $\models$ denotes the logical consequence and $\bot$ denotes *false*.

Definition 3.2 (LFI)**.** A *Learning from Interpretations* (ILP$_{LFI}$) task T is a tuple $\langle B, S, E^+, E^- \rangle$ where B is a definite clausal theory, S is a set of clauses and each element of E^+ and E^- is a set of facts. A hypothesis $H \subseteq S$ is an *inductive solution* of T if and only if:

- $\forall e^+ \in E^+ : M(B \cup \{e^+\})$ satisfies H,
- $\forall e^- \in E^- : M(B \cup \{e^-\})$ does not satisfy H.

where $M(\cdot)$ is the minimal model of $(\cdot)$.

This means that, if we do not have a background knowledge, each positive example must be a model of H, and no negative example should be a model of H.

Definition 3.3 (LFS)**.** A *Learning from Satisfiability* (ILP$_{LFS}$) task T is a tuple $\langle B, S, E^+, E^- \rangle$ where B is a clausal theory, S is a set of definite clauses and E^+ and E^- are sets of clausal theories. A hypothesis $H \subseteq S$ is an inductive solution of T if and only if:

- $\forall e^+ \in E^+ : B \cup H \cup \{e^+\}$ has at least one model,
- $\forall e^+ \in E^- : B \cup H \cup \{e^-\}$ has no models.

In [16] is proved that ILP$_{LFS}$ can simulate ILP$_{LFE}$ using the fact that $P \models e$ if and only if $P \cup \{\neg e\}$ has no models.

3.2. Learning from Answer Set

We focus on the approaches to ILP by using Answer Set semantics. We will now switch from classical logic to logic of *stable model* [17]. The first thing to notice is that in ASP there can be zero or many answer sets of a program. For this reason we talk about *brave entailment* ($\models_b$) and *cautious entailment* ($\models_a$): an atom a is bravely entailed by a program P if and only if at least one answer set P contains a, while it is cautiously entailed if every answer set contains it. We will denote with $AS(P)$ the set of all answer sets of P.

Definition 3.4. A *cautious induction* (ILP$_c$) task T is a tuple $\langle B, S, E^+, E^- \rangle$ where B is an ASP program, S is a set of ASP rules and E^+ and E^- are sets of ground atoms. A hypothesis $H \subseteq S$

is an inductive solution of T if and only if $AS(B \cup H) \neq \emptyset$ and $\forall A \in AS(B \cup H), E^+ \subseteq A$ and $E^- \cap A = \emptyset$.

This means that all examples should be covered in *every* answer set and that $B \cup H$ should be satisfiable. On the other hand, brave induction require all the examples to be covered by *at least one* answer set.

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

$$B = \begin{cases} \texttt{dog(X):- alano(X).} \\ \texttt{dog(X):- corgi(X).} \\ \texttt{alano(Scooby).} \\ \texttt{corgi(Muick).} \end{cases} \qquad S = \begin{cases} h_1: \ \texttt{long\_legs(X):- dog(X).} \\ h_2: \ \texttt{long\_legs(X):- dog(X), not corgi(X).} \\ h_3: \ \texttt{0\{long\_legs(X)\}1:- dog(X).} \\ h_4: \ \texttt{0\{long\_legs(X)\}1:- dog(X),} \\ \qquad \texttt{not corgi(X).} \end{cases}$$

$E^+ = \{\texttt{long\_legs(Scooby)}\}$
$E^- = \{\texttt{long\_legs(Muick)}\}$

The inductive solution ILP_c(T) is h_2 because $B \cup \{h_2\}$ has exactly one answer set, and it contains `long_legs(Scooby)` *but not* `long_legs(Muick)`.

Definition 3.5. A *brave induction* (ILP$_b$) task T is a tuple $\langle B, S, E^+, E^- \rangle$ where B is an ASP program, S is a set of ASP rules and E^+ and E^- are sets of ground atoms. A hypothesis $H \subseteq S$ is an inductive solution of T if and only if $\exists A \in AS(B \cup H)$ such that $E^+ \subseteq A$ and $E^- \cap A = \emptyset$.

Example 2. *Consider the ILP_b task $T = \langle B, S, E^+, E^- \rangle$ where B, S, E^+, E^- are the same as ones of example 1. $\{h_2\}$, $\{h_3\}$, $\{h_4\}$ are solutions of ILP_b(T) as each of $AS(B \cup \{h_2\})$, $AS(B \cup \{h_3\})$ and $AS(B \cup \{h_4\})$ contains* `long_legs(Scooby)` *but not* `long_leg(Muick)`.

The problem of brave induction is that it cannot learn constraints because it is not able to reason about what is true in *every* answer set. To overcome this, ILASP uses *Induction of Stable Models* which allows to set conditions over multiple answer sets. The idea is to use partial interpretations as examples.

Definition 3.6. 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 $e^{exc} \cap I = \emptyset$.

Definition 3.7. An *induction of stable models* (ILP$_{sm}$) task T is a tuple $\langle B, S, E \rangle$, where B is an ASP program, S is the hypothesis space and E is a set of example partial interpretations. A hypothesis H is an inductive solution of T if and only if $H \subseteq S$ and $\forall e \in E, \exists A \in AS(B \cup H)$ such that A extends e.

Keeping in mind all these definitions, we can finally declare what is Learning from Answer Sets. In particular we are presenting the learning framework used by ILASP to infer new constraints.

Definition 3.8. A *Learning from Answer Sets* task is a tuple $T = \langle B, S, E^+, E^- \rangle$ where B is an ASP program, S a set of ASP rules and E^+ and E^- are finite sets of partial interpretations. A hypothesis $H \subseteq S$ is an inductive solution of T if and only if:

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

Since positive examples have to be extended by at least one answer set, they can be considered as characterised by brave induction, while negative examples as characterised by cautious induction, as we want them to not be extended by any answer set. This is what ILASP use to infer new rules.

3.3. ILASP Algorithm

ILASP uses a *Conflict-Driven ILP* (CDILP) process which iteratively constructs, for each example, a set of *coverage constraints*.

Definition 3.9. Let S be a rule space. A *coverage formula* over S takes one of the following forms:

- R_{id}, for some $R \in S$,
- $\neg F$, where F is a coverage formula over S,
- $F_1 \vee ... \vee F_n$, where $F_1, ...F_n$ are coverage formulas over S,
- $F_1 \wedge ... \wedge F_n$, where $F_1, ...F_n$ are coverage formulas over S.

The semantics of coverage formulas are defined as follows. Given a hypothesis H:

- R_{id} accepts H if and only if $R \in H$,
- $\neg F$ accepts H if and only if F does not accept H,
- $F_1 \vee ... \vee F_n$ accepts H if and only if $\exists i \in$ [1,n] s.t. F_i accepts H
- $F_1 \wedge ... \wedge F_n$ accepts H if and only if $\forall i \in$ [1,n] s.t. F_i accepts H

Definition 3.10. A *coverage constraint* is a pair $\langle e, F \rangle$ where e is an example in E and F is a coverage formula, such that for any $H \subseteq S$, if e is covered then F accepts H.

At every iteration, an optimal hypothesis H^* (i.e. the shortest), that is consistent whit the existing coverage constraints, is computed. Then the *counterexample search* starts. If an example that is not covered by H^* exists then a new coverage constraint for that one is generated. Then the *conflict analysis* takes place which compute a coverage formula F that does not accept H but that must hold for the most recent counterexample to be covered. If the counterexample does not exist then H^* is returned as the optimal solution of the task.

3.4. ILASP Syntax

ILASP syntax for background knowledge is the same as the ASP one except for the fact that it is not allowed to define constants. Concerning examples E^+ e E^-, they are declared as follows: `#pos({e`^{inc}`},{e`^{exc}`})` and `#neg({e`$^{-}$`},{})`. For negative example we always leave an empty set because we do not have the partition between inclusions and exclusions as happens for positive ones. For example, if we want to express E^+ and E^- of example 1, we

have: `#pos({long_legs(Scooby)},{})` and `#neg({long_legs(Muick)},{})`. In this case we do not have exclusions for positive examples so the set is left empty.

To describe hypothesis spaces we have two way. The first one consists of mode declarations. A mode declaration is a ground atom whose arguments can be placeholders. They are terms like `var(t)` or `const(t)` for some constant term `t`. The idea is that these can be replaced respectively by any variable or constant of type `t`. Their syntax is the following, where `pred` stands for a generic predicate and (·) would be replaced with `var(t)` or `const(t)`:

- to declare what can be in the head: `#modeh(pred(·))` or `#modeha(pred(·))` if we want to allow aggregates,
- to declare what can be in the body: `#modeb(n, pred(·))` where `n` in the maximum times `pred(·)` can appear in the body
- to declare conditions: `#modec(·)`, where the argument can be, for example, `var(t)>var(t)`.

The second way is to directly declare the rules we want to include in the hypothesis space.

4. Puzzles

In this section we present some classical puzzles, and a standard declarative encoding in ASP. We will compare these encoding with those generated by ILASP in Section 5 and also to use some predicates in the hypothesis search space. The full encodings can be found here. Every puzzle uses a $n \times n$ board that is considered as a matrix so that $(1, 1)$ is the top-left cell and (n, n) is the bottom-right one.

4.1. N-queens

N-queens is a well known puzzle in which we have a $n \times n$ board. The aim is to place n queens such that they cannot attack each other; in other words, two queens cannot stay on the same row, column or diagonal. `queen(X, Y)` is defined to describe that a queen is on the board in the position `(X, Y)`. To completely model the game we need to state which reciprocal positions are not allowed between queens and that there must be n queens.

```
1  n{queen(X,Y): coord(X), coord(Y)}n.
2  same_col(X1,Y,X2,Y) :- queen(X1,Y), queen(X2,Y), X1!=X2.
3  same_row(X,Y1,X,Y2) :- queen(X,Y1), queen(X,Y2), Y1!=Y2.
4  same_diagonal(X1,Y1,X2,Y2) :- queen(X1,Y1), queen(X2,Y2), X1!=X2, |X1-X2|=|Y1-Y2|.
5  :- same_row(X1,Y1,X2,Y2).
6  :- same_col(X1,Y1,X2,Y2).
7  :- same_diagonal(X1,Y1,X2,Y2).
```

4.2. Star Battle

Star Battle is a logic puzzle whose goal is to find a way to put $s \cdot n$ stars on a $n \times n$ board (where s is the number of allowed stars per row, column and field). The board is divided in n fields of different size and shape and the stars must be placed satisfying the following constraints: two stars cannot be adjacent horizontally, vertically or diagonally, and there must be s stars on each row, column and field. An example is shown in Figure 1.

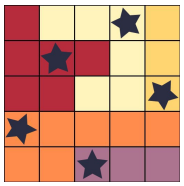

Figure 1: Example of a solved board (s=1); different colours define different fields.

Since boards can differ from one to another by fields, `cell_in_field(C, F)` is used to describe that cell `C` belongs to the field `F`. Moreover, `neighbour(C1,C2)` was used to express that a cell `C2` is adjacent (even diagonally) to `C1`.

```
neighbour((X1,Y1), (X2,Y2)) :- cell((X1,Y1)), cell((X2,Y2)),
↪  |X1-X2|+|Y1-Y2|=2, X1!=X2, Y1!=Y2.
neighbour(C1, C2) :- adj(C1, C2).
adj((X1,Y1), (X2,Y2)) :- cell((X,Y1)), cell((X,Y2)), |Y1-Y2|+|X1-X2|=1.
```

Finally, we express the constraints for the placement of stars:

```
s{star_in_cell((X,Y)): cell((X,Y))}s :- 1<=X, X<=n.
s{star_in_cell((X,Y)): cell((X,Y))}s :- 1<=Y, Y<=n.
s{star_in_cell(C): cell_in_field(C, F)}s :- field(F).
:- star_in_cell(V1), star_in_cell(V2), neighbour(V1,V2), V1!=V2.
```

For simplicity in next sections we will always consider the case where s=1.

4.3. Flow Lines

Flow Lines consists of a board where are placed some pairs of dots. The goal is to connect the corresponding dots of every pair while filling the entire board (as shown in Figure 2). The problem has been solved considering that each pair of dots can be seen as a pair of start and end of a path. For this reason, we encoded `start(C, P)` and `end(C, P)` as facts. The predicate `cell_in_path(C, P)` describes that cell `C` belongs to path `P`; each cell of the board has to belong to exactly one path.

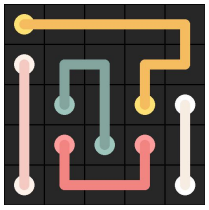

Figure 2: Example of a solved board; each circle is a start or a end

```
1{cell_in_path(C, P): path(P)}1 :- cell(C).
```

To describe the path, for each cell is defined a `next(C1, C2)` and a `prev(C1, C2)` stating, respectively, that the next step after `C1` in its path is `C2` and that the previous step for `C1` in its path was `C2`.

```
2  1{next(C,C1): cell_in_path(C1,P), adj(C,C1)}1 :- cell_in_path(C,P), not end(C,P).
3  1{prev(C,C1): cell_in_path(C1,P), adj(C,C1)}1 :- cell_in_path(C,P), not start(C,P).
```

The path we want to find is the one connecting start to end without crossing other paths.

```
4  crossing(P1, P2) :- cell_in_path(C, P1), cell_in_path(C, P2), P1!=P2.
```

To conclude we state what cannot happen.

```
5  :- start(C, P), end(C, P).                    % start and end cannot be coincident
6  :- crossing(P, P1), P!=P1, path(P), path(P1).  % paths cannot cross
7  :- start(C1, P), start(C2, P), C1!=C2, path(P). % each path has only one start
8  :- end(C1, P), end(C2, P), C1!=C2, path(P).    % each path has only one end
```

4.4. 15 Puzzle

The last example we considered is the 15 puzzle, a sliding puzzle having 15 square tiles numbered from 1 to 15 in a grid 4×4, leaving one unoccupied tile position. Note that the puzzle is a planning problem because we are searching a sequence of actions that could lead to the solution. In fact, the aim is to move the tiles, using the empty space, until they are ordered from 1 to 15 starting from the top left corner. The solution proposed uses a slightly different definition of adjacency from the other games.

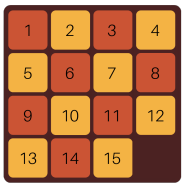

Figure 3: Solved 15 puzzle board.

Indeed, we only need to know tiles adjacent to the empty space. The effects of a move are described by `move(N,S)`, meaning that the tile with number `N` has been moved at time `S`.

```
1  adj_empty(N, S) :- value(X1, Y1, S, N), value(X2, Y2, S, n*n), coord(X1;X2;Y1;Y2), step(S),
   ↪  card(N), |X1-X2|+|Y1-Y2|=1, N!=n*n.
2  :- move(N,S), move(M,S), M!=N, card(N), card(M), step(S).
3  :- move(N,S), not adj_empty(N,S), card(N), step(S).
4  move(N,S) :- value(X,Y,S,N), value(XE,YE,S,n*n), value(X,Y,S+1,n*n), value(XE,YE,S+1,N),
   ↪  adj_empty(N,S), coord(X;Y;XE;YE), step(S;S+1), card(N), N!=n*n.
5  value(X,Y,S+1,n*n) :- move(N,S), value(X,Y,S,N), card(N), coord(X;Y), step(S;S+1).
6  value(X,Y,S+1,N) :- move(N,S), value(X,Y,S,n*n), card(N), coord(X;Y), step(S;S+1).
```

We added some more constraints. First, we want to have a move at each step until the solution is reached, i.e. there is no step with no move before the solution is reached. Secondly, if a tile `N` has not been moved then it will stay in the same position. Furthermore, if tile `N` has been moved at time `S`, the move of the next step must be on another tile as we do not want to go back to the previous configuration.

```
7  1{move(M,S-1): card(M), step(S-1)}1 :- move(N,S), S>1, card(N), step(S).
8  value(X,Y,S+1,N) :- not move(N,S), value(X,Y,S,N), coord(X;Y), step(S;S+1), card(N), N!=n*n.
9  :- move(N,S), move(N,S+1), step(S+1), step(S).
```

Finally, to reach the solution, we say that a board is not ordered if there is at least one tile in the wrong place and then we force the board to be ordered at `max` timestep. `max` is, indeed, the maximum number of steps needed to solve any 15 puzzle (that is 80 for 4×4 version, while for 3×3 version it is 31). By imposing such a bound on the number of steps, the solution may not be optimal, but this is not an essential aspect of our study.

```
10 not_ordered(S) :- value(X,Y,S,N), coord(X;Y), step(S), card(N), N!=n*(X-1)+Y.
11 :- not_ordered(max).
```

5. Examples and Hypotheses

In this section we are going to explain the examples that we used for each puzzle. In order to make ILASP to learn constraints, from our ASP modelling we kept, as background knowledge, only the definition of predicates.
All data reported refers on tests done on a NVIDIA GeForce RTX 3060 (16GB).

5.1. N-queens

For n-queens puzzle it was sufficient to consider one positive and one negative example for each constraint we wanted to learn: a powerful aspect of ILASP is that it can generalise from very few examples, making it possible to learn complex knowledge without needing large datasets.

```
1 %% in every column there cannot be more than one queen
2 #pos({queen(1,3)}, {queen(2,3), queen(3,3), queen(4,3)}}).
3 #neg({queen(4,3), queen(2,3)},{}).
4 %% in every row there cannot be more than one queen
5 #pos({queen(1,3)}, {queen(1,2), queen(1,4), queen(1,1)}}).
6 #neg({queen(2,1), queen(2,4)},{}).
7 %% in every diagonal there cannot be more than one queen
8 #pos({queen(2,4)}, {queen(3,3), queen(4,2), queen(1,3)}}).
9 #neg({queen(1,1), queen(4,4)},{}).
```

We also defined a search space S to speed up the algorithm, that is:

```
10  :- same_row(X1, Y1, X2, Y2).          % only one queen per row
11  :- same_col(X1, Y1, X2, Y2).          % only one queen per col
12  :- same_diagonal(X1, Y1, X2, Y2).     % only one queen per diagonal
13 1{queen(X,Y): coord(Y)}1 :- coord(X). % only one queen per row
14 1{queen(X,Y): coord(X)}1 :- coord(Y). % only one queen per col
15 4{queen(X,Y): coord(X), coord(Y)}4.    % four queens in total
```

With this set of examples the program was able to learn the constraints at line 11, 12 and 13. Time required to reach this solution, with 6 rules in search space and 6 examples (3 positive and 3 negative) are:

```
Pre-processing                    : 0.007s
Hypothesis Space Generation       : 0.004s
Conflict analysis                 : 0.006s
```

```
Counterexample search                 : 0.008s
Hypothesis Search                     : 0.021s
                                Total : 0.047s
```

Note that it learns line 13 instead of 10 because line 13 is needed to generate answer sets containing istances of `queen(X, Y)`. The others constraints in S are discarded even if they are correct because ILASP returns the *minimum* set. The minimum set consists of the minimum number of constraints that the program has to learn to cover all examples. Given any framework ILP_X, learning task T and example e, we say that e is covered by a hypothesis if the hypothesis meets all conditions that T imposes on e. If two or more rules in the hypothesis space are equivalent, ILASP will chose the smallest in terms of number of atoms.

5.2. Star Battle

In the following examples we omitted those representing the configuration where we have more than one star per row/column or stars that are adjacent as they are similar to the n-queen's ones. Moreover, examples are defined also by a third argument that is context. Context is a set of ground atoms that enrich the knowledge base only for that single example.

```
1 %% two stars cannot stay in the same field
2 #pos({star_in_cell((1,4))},
3      {star_in_cell((1,3)), star_in_cell((2,4)), star_in_cell((3,4)), star_in_cell((4,4))},
4      {cell_in_field((1,1),1). cell_in_field((1,2),1). cell_in_field((1,3),2).
       ↪ cell_in_field((1,4),2). cell_in_field((2,1),1). cell_in_field((2,2),1).
       ↪ cell_in_field((2,3),1). cell_in_field((2,4),2). cell_in_field((3,1),1).
       ↪ cell_in_field((3,2),3). cell_in_field((3,3),3). cell_in_field((3,4),2).
       ↪ cell_in_field((4,1),4). cell_in_field((4,2),4). cell_in_field((4,3),4).
       ↪ cell_in_field((4,4),2).}).
5 #neg({star_in_cell((1,2)),star_in_cell((2,3))}, {},
6      {cell_in_field((1,1),1). cell_in_field((1,2),1). cell_in_field((1,3),2).
       ↪ cell_in_field((1,4),2). cell_in_field((2,1),1). cell_in_field((2,2),1).
       ↪ cell_in_field((2,3),1). cell_in_field((2,4),2). cell_in_field((3,1),1).
       ↪ cell_in_field((3,2),3). cell_in_field((3,3),3). cell_in_field((3,4),2).
       ↪ cell_in_field((4,1),4). cell_in_field((4,2),4). cell_in_field((4,3),4).
       ↪ cell_in_field((4,4),2).}).
```

The search space S considered this time is:

```
7 1{star_in_cell(C): cell_in_field(C, F)}1 :- field(F).
```

Then we enriched it using mode declarations. The set of rules produced by mode declarations can be expensive in terms of cost so we introduced the predicate `incompatible(C1,C2)` which is inferred by any of `same_row`, `same_col`, `same_field` or `same_neighbour`.

Example 3. *With just the following mode declarations, which will also generate the rules we are interested in, the search space will contain more than 56000 rules.*

```
#modeha(star_in_cell(var(cell))).
```

```
#modeb(1, same_col(var(cell), var(cell))).
#modeb(1, same_row(var(cell), var(cell))).
#modeb(1, same_field(var(cell), var(cell))).
#modeb(1, neighbour(var(cell), var(cell))).
#modeb(1, field(var(field))).
#modeb(2, var(cell)!=var(cell)).
```

Running ILASP with this configuration will end in process killing itself due to exceeding memory.

```
8  #modeha(star_in_cell(var(cell))).   % generate rule 12
9  #modeb(2, star_in_cell(var(cell))).
10 #modeb(1, incompatible(var(cell), var(cell)), (symmetric)).
```

The code above is used to create hypotheses and the search space it generate includes 526 rules. In particular line 8 says that we can produce rules where the head could contain `star_in_cell(C)`, where C is a variable `cell(C)`, as aggregate. Line 2 and 3 states that body rules can show up to two `star_in_cell(C)` and up to one `incompatible(C1,C2)` (symmetric means that rules with arguments `(C2,C1)` are considered equivalent to the ones with `(C1,C2)`). The program was able to infer:

```
11  :- star_in_cell(V1); star_in_cell(V2); incompatible(V1,V2). % no incompatibility
12 0{star_in_cell(V1)}1 :- incompatible(V1,V2). % star_in_cell should appear in some AS
```

Running ILASP with a search space of size 526 and 9 examples (2 positive, 7 negative) is:

```
Pre-processing                          : 0.006s
Hypothesis Space Generation             : 0.041s
Conflict analysis                       : 0.055s
Counterexample search                   : 0.017s
Hypothesis Search                       : 2.596s
                                  Total : 2.718s
```

Note that rule at line 12 seems useless but, as in the background knowledge we never talked about `star_in_cell`, it would be in no answer set and so any example containing `star_in_cell` would not be covered. Adding this constraint, answer sets containing `star_in_cell` are admitted and so examples can be covered. If we remove line 8, that is the one generating the rule in line 12, ILASP will learn:

```
13  1{star_in_cell(C): cell_in_field(C,F)}1 :- field(F).
```

instead of the previous. In this case, when both rules 11 and 12 belong to S, the second one is chosen because it is shorter in terms of atoms contained. The problem with the latter is that, if used instead of 11, it admits also solutions where stars are less than n. If we remove:

```
14 1{cell_in_field(C1, F): field(F)}1 :- cell(C1). % removed from B
```

and then put it in S, it will learn again 11 and not 14.

5.3. Flow Lines

Flow lines has the most numerous set of examples in this paper. Furthermore, as for Star Battle, another predicate, called `not_good`, was introduced. It simply means that if `not_good(P)` then path `P` is not valid for some reason such as crossing another path or being disconnected as shown in Figure 4. The most relevant examples are:

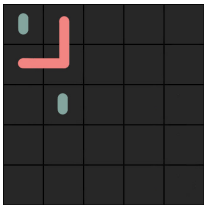

Figure 4: Graphical representation of last negative example.

```
1  %% each path has only one start
2  #pos({start((1,2),1), cell_in_path((1,2),1)}, {start((2,3),1)}).
3  %% start and end cannot be coincident
4  #neg({start((1,1),1), end((1,1),1)},{}).
5  %% next and previous of a cell cannot be the same
6  #pos({next((2,2),(3,2))}, {prev((2,2),(3,2))}).
7  #neg({next((2,2),(3,2)), prev((2,2),(3,2))},{}).
8  %% previous and next must be in the same path
9  #pos({prev((2,1),(3,1)), cell_in_path((2,1),3)}, {cell_in_path((3,1),4)}).
10 #pos({next((2,1),(3,1)), cell_in_path((2,1),3)}, {cell_in_path((3,1),4)}).
11 #neg({next((2,2),(3,2)), cell_in_path((2,2),1), cell_in_path((3,2),4)},{}).
12 %% previous of a cell cannot be the previous of another cell
13 #neg({prev((3,1),(1,1)), prev((2,1),(1,1))},{}).
14 %% every cell of a path must be reachable from start
15 #neg({cell_in_path((2,1),1), cell_in_path((2,2),1), cell_in_path((1,2),1),
   ↪  cell_in_path((1,1),2), start((3,2),2)},{}).
```

With all of them, ILASP is able to learn the following:

```
16  :- not_good(P).
17  :- end(C, P); start(C, P).
18 1 {start(C, P) : cell(C) } 1 :- path(P).
19 1 {end(C, P) : cell(C) } 1 :- path(P).
20  :- prev(C1, C2); P1 != P2; cell_in_path(C2, P2); cell_in_path(C1, P1).
```

Actually, the search space for hypothesis we gave included also these rules:

```
21  :- crossing(P1, P2).                          %redundant
22  :- next(C, C1), prev(C, C1).
23  :- cell(C), not path_to_start(C).
24 cell_in_path(C, P) :- start(C, P).
25  :- next(C1, C), next(C2, C), C1!=C2.
26  :- prev(C1, C), prev(C2, C), C1!=C2.          %redundant
27  :- cell_in_path(C, P1), cell_in_path(C, P2), P1!=P2.
28  :- cell_in_path(C1, P1), cell_in_path(C2, P2), P1!=P2, next(C1, C2).  %redundant
```

Some of them are redundant such as those in line 21 and line 26 (because of `not_good(P):-crossing(P1,P2),P1!=P2`) while others, such as the one in line 28, seem to be missing even if there are negative example where there are cells that cannot be reached from start (see Figure 4) but this only means that they are not necessary to cover all examples. With 12 rules in the search space and 13 examples (6 positive and 7 negative), the amount of time needed to run ILASP on this puzzle is:

```
Pre-processing                    : 0.006s
Hypothesis Space Generation       : 0.004s
Conflict analysis                 : 0.037s
Counterexample search             : 0.177s
Hypothesis Search                 : 0.681s
                            Total : 0.905s
```

5.4. 15 Puzzle

15 puzzle examples shown below are just the most relevant. We considered the case of a grid 3×3 instead of the original one to reduce the execution time required by the algorithm. In a 3×3 board the tiles are numbered from 1 to 8 (in ASP program we use 9 to describe the empty slot). Moreover, to make ILASP learn more interesting rules, we changed the background knowledge by modifying the rule `move(N,S)`. In fact, we removed `adj_empty(N,S)` from its body.

```
1  %% at each step only one move is permitted
2  #pos({move(4,1)},{move(5,1)}).
3  #neg({move(4,2), move(1,2)},{}).
4  %% if N was moved at time S then I will not move it again at S+1
5  #pos({move(3,5)},{move(3,6)}).
6  %% each number appears only one time per step
7  #neg({value(3,1,5,3), value(1,2,5,3)},{},{}).
8  %% moving tiles not adj to the empty one is not permitted
9  #pos({move(5,1), adj_empty(5,1)},{}).
10 #neg({move(1,1),adj_empty(2,1),adj_empty(4,1),adj_empty(8,1),adj_empty(6,1)},{}).
11 %% reaching the solution
12 #pos({not_ordered(30)},{not_ordered(31)}).
13 #neg({not_ordered(31)},{}).
```

The search space in this case it is quite vast.

```
14  :- not_ordered(31).
15  :- not move(9, S), step(S).
16  :- value(X1, Y1, S, N), value(X2, Y2, S, N), X1!=X2.
17  :- value(X1, Y1, S, N), value(X2, Y2, S, N), Y1!=Y2.
18  1{move(N,S): card(N), step(S)}30.
19  :- move(N, S), move(N, S+1), step(S+1), step(S).
20  :- move(N, S), not adj_empty(N, S), card(N), step(S).
21  :- not not_ordered(S), not_ordered(S+1), step(S), step(S+1).
22  :- move(N, S), not adj_empty(N, S), card(N), step(S).
23  :- not_ordered_row(X, S), not not_ordered_row(X+1, S), coord(X), coord(X+1).
24 1{value(X, Y, S, N): card(N)}1 :- step(S), coord(X), coord(Y).
25 1{value(X, Y, S, N): coord(X), coord(Y)}1 :- card(N), step(S).
26  :- not not_ordered(S), move(N, S+1), step(S), step(S+1), card(N).
```

```
27   :- step(S), not not_ordered(S), move(N,S1), S1>S, step(S1), card(N).
28   :- move(N, S), move(M, S), M!=N, card(N), card(M), step(S).
29  1{move(M, S-1): card(M), step(S-1)}1 :- move(N, S), S>1, card(N), step(S).
```

The solver was able to learn only the following:

```
30   :- step(S); card(M); card(N); M != N; move(M, S); move(N, S).
31   :- step(S); step(S+1); move(N, S+1); move(N, S).
32  1 {move(N, S) : step(S), card(N) } 30.
```

As was happening in Star Battle, if we do not add to the search space also the following rule, there will be answer sets that do not contain instantiations of `adj_empty` and so our examples will not be covered.

```
33  0{adj_empty(N, S): card(N)}4 :- step(S).
```

Now running ILASP we get the additional rules:

```
34  2 {adj_empty(N, S) : card(N) } 4 :- step(S).
35    :- move(V1,V2); not adj_empty(V1,V2).
```

Finally, to get it learn the goal of the game we have to add to S the following rule:

```
36  0{not_ordered(S): step(S)}30.
```

That permits to include in the solution also:

```
37   :- not_ordered(31).
```

Time required, with a search space of 19 and 20 examples (11 positive and 9 negative), was:

```
Pre-processing                    : 0.006s
Hypothesis Space Generation       : 0.004s
Conflict analysis                 : 0.22s
Counterexample search             : 0.707s
Hypothesis Search                 : 35.94s
                            Total : 36.878s
```

We tried to use more general examples in order to learn these last constraints, such as the ones below, but unfortunately they were not enough. With general examples, we mean complete solutions generated by the ASP program described in section 4.4 used as positive examples with no exclusions. Those example are general in the sense that they are not constructed to cover specific constraints. The reason is that in a real life problem, we do not always know what is permitted and what is not and so we cannot always create examples that describe a precise behaviour.

```
38 #pos({move(6,13), move(5,12), move(4,11), move(7,10), move(8,9), move(4,8), move(5,7),
   ↪ move(6,6), move(4,5), move(8,4), move(7,3), move(5,2), move(6,1)},
39      {not_ordered(14), not_ordered(17), not_ordered(20), not_ordered(23), not_ordered(26),
        ↪ not_ordered(29), not_ordered(15), not_ordered(18), not_ordered(21), not_ordered(24),
        ↪ not_ordered(27),not_ordered(30), not_ordered(16), not_ordered(19), not_ordered(22),
        ↪ not_ordered(25), not_ordered(28), not_ordered(31)},
40      {value(1,1,1,1). value(1,2,1,2). value(1,3,1,3). value(2,1,1,5). value(2,2,1,6).
        ↪ value(2,3,1,9). value(3,1,1,7). value(3,2,1,8). value(3,3,1,4).}).
41 #neg({value(1,1,28,1), value(1,2,28,2), value(1,3,28,3), value(2,1,28,4), value(2,2,28,5),
   ↪ value(2,3,28,6), value(3,1,28,7), value(3,2,28,8), value(3,3,28,9), not_ordered(30)},{}).
```

Table 1 reports a little summary of what we got.

6. Conclusions and Future Works

In this article, we presented a way to describe some puzzles in ILASP focusing on the effects that different examples have on the solution. That exploration led us to get good results, learning all the rules needed to solve puzzles, except for Flow Lines. What emerged is the difficulty to learn from very general examples. Indeed, all the examples used were constructed knowing what we wanted to learn: when we tried, for example in 15 puzzle, to give it generic examples of solutions, it was not able to infer new rules. Moreover, we always gave a hypothesis space containing correct rules for the problem, even if redundant. Using mode declarations worked too but it is very space consuming and so not practical when trying to learn complex and long constraints. Lastly, as the Definition 3.8 states, positive examples must appear in at least one answer set and so we always need a rule that forces the generation of the instatiation of the predicates for which we want to learn a rule about. If we do not, ILASP will return UNSATISFIABLE even if the examples given are correct.
As future work, we would like to better explore the functionality of ILASP and how to improve or enrich them. In fact, ILASP can be used also with *weak* constraints and taking account of *preference ordering* of constrains, not discussed in this paper. Furthermore, future studies could interest the use of parallel computation in order to reduce time and space required to explore bigger hypothesis space.

	N-Queens	Star Battle	Flow Lines	15 Puzzle
Background Knowledge	Rules defining when queens are on same row, column or diagonal	Rules defining when stars are on same row, column, field or when are adjacent	Rules defining what is a path between start and end cells, what means crossing one path with another and, in general, what is not an admitted path	Rules defining what is a move (without adjacency constraint) and what means that the board is not ordered
Examples	One positive and one negative for each constraint to be learnt	One positive and one negative for each constraint to be learnt	More than one positive and one negative for each constraint to be learnt	One positive and/or one negative for each constraint to be learnt
Hypothesis Space	All constraints necessary to solve the puzzle	All constraints necessary to solve the puzzle (considering also a constraint to include `star_in_cell` in answers sets)	All constraints necessary to solve the puzzle	All constraints necessary to solve the puzzle, plus some alternatives for the same constraints
Learnt	All constraints in the hypothesis space	Minimum set of constraints in the hypothesis space showing how to place stars	Constraints telling that is not possible to cross paths, to do not have exactly one start and one end per path and to have previous and next cells in different paths	Minimum set of constraints in the hypothesis space to solve the puzzle
Not Learnt	\	Rule telling there must be exactly n stars	Redundant rules and rule saying that a path cannot have "turnarounds"	Redundant rules

Table 1
Summary of ILASP results

References

[1] F. Zelezny, N. Lavrac, Inductive logic programming, Machine learning 76 (2009).

[2] W. Ding, M. Abdel-Basset, H. Hawash, A. M. Ali, Explainability of artificial intelligence methods, applications and challenges: A comprehensive survey, Information Sciences (2022).

[3] A. Cropper, S. Dumančić, R. Evans, S. H. Muggleton, Inductive logic programming at 30, Machine Learning (2022) 1–26.
[4] M. Law, Inductive learning of answer set programs, Ph.D. thesis, University of London, 2018.
[5] N. Rizzo, A. Dovier, 3coSoKu and its declarative modeling, J. Log. Comput. 32 (2022) 307–330. doi:10.1093/logcom/exab086.
[6] L. Cian, T. Dreossi, A. Dovier, Modeling and solving the rush hour puzzle, in: R. Calegari, G. Ciatto, A. Omicini (Eds.), Proceedings of the 37th Italian Conference on Computational Logic, Bologna, Italy, June 29 - July 1, 2022, volume 3204 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 294–306.
[7] N. Zhou, A. Dovier, A tabled prolog program for solving sokoban, Fundam. Informaticae 124 (2013) 561–575. doi:10.3233/FI-2013-849.
[8] S. Muggleton, A. Paes, V. Santos Costa, G. Zaverucha, Chess revision: Acquiring the rules of chess variants through fol theory revision from examples, in: Inductive Logic Programming: 19th International Conference, ILP 2009, Leuven, Belgium, July 02-04, 2009. Revised Papers 19, Springer, 2010, pp. 123–130.
[9] S. Legras, C. Rouveirol, V. Ventos, The game of bridge: A challenge for ilp, in: Inductive Logic Programming: 28th International Conference, ILP 2018, Ferrara, Italy, September 2–4, 2018, Proceedings 28, Springer, 2018, pp. 72–87.
[10] M. Law, A. Russo, K. Broda, The ilasp system for inductive learning of answer set programs, arXiv preprint arXiv:2005.00904 (2020).
[11] D. Meli, M. Sridharan, P. Fiorini, Inductive learning of answer set programs for autonomous surgical task planning: Application to a training task for surgeons, Machine Learning 110 (2021) 1739–1763.
[12] F. Daniele, M. Filippo, R. Luca, S. Matteo, F. A. D'Asaro, et al., Using inductive logic programming to globally approximate neural networks for preference learning: challenges and preliminary results, in: CEUR WORKSHOP PROCEEDINGS, 2022, pp. 67–83.
[13] F. A. D'Asaro, M. Spezialetti, L. Raggioli, S. Rossi, Towards an Inductive Logic Programming Approach for Explaining Black-Box Preference Learning Systems, in: Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, 2020, pp. 855–859. URL: https://doi.org/10.24963/kr.2020/88. doi:10.24963/kr.2020/88.
[14] S. Calo, I. Manotas, G. de Mel, D. Cunnington, M. Law, D. Verma, A. Russo, E. Bertino, AGENP: An ASGrammar-based GENerative Policy Framework, Springer International Publishing, Cham, 2019, pp. 3–20. URL: https://doi.org/10.1007/978-3-030-17277-0_1. doi:10.1007/978-3-030-17277-0_1.
[15] S. Muggleton, Inductive logic programming, New generation computing 8 (1991) 295–318.
[16] L. De Raedt, Logical settings for concept-learning, Artificial Intelligence 95 (1997) 187–201.
[17] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming., in: ICLP/SLP, volume 88, Cambridge, MA, 1988, pp. 1070–1080.