Inductive Logic Programming with Andante

Simon Jacquet

August 30, 2022

0 Introduction

This document is intended as documentation for the Andante python package. It makes the link between Inductive Logic Programming (ILP) theory and its implementation in the Andante toolbox.

All jupyter notebook examples can be found in the DOCUMENTATION EXAMPLES.IPYNB file in the git-repository for the Andante package at the address: `https://gitlab.unamur.be/sijacque/andante`

Section 1 gives the terminologies used throughout the rest of the document and in the implementation.

Section 2 gives an overview of Logic Programming. In particular, it describes deduction in section 2.1. Also, it describes how solving works in Logic Programming (section 2.2).

Section 3 dwelves into Inductive Logic Programming. It describes induction in section 3.1, introducing notions such as substitution (section 3.1.1) and clause subsumption (section 3.1.2). Section 3.2 gives an overview of all induction-related algorithms. Finally, section 3.3 gives a few words on the Andante solver.

Section 4 introduces the Andante interface that allows for good understanding of the ILP learner described in the previous section.

At last, section 5 provides the structure of the Andante package as a whole. It regroups all modules in disctinct groups depending on their function.

1 Terminology

The terminologies used in the Andante package come from various resources: first order logic, prolog and progol. Some of those use the same terms but with different meaning. This section aims to clarify the notions used.

Section 1.1 defines all terms used in the context of Logic Programming, whereas section 1.2 defines those used in Inductive Logic Programming.

1.1 Logic Programming concepts

Logic Programming inscribes itself in the first order logic framework. The most well-known Logic Programming language is prolog. Alas, the terminoloy is not consistent in both framework. Here is described the terminology used in the Andante toolbox.

Variable: Universally defined variable. Implemented as ANDANTE.LOGIC_CONCEPTS.VARIABLE.

```
<Variable> ::= [A-Z] [a-zA-Z_0-9]*
```

Function: Operator that maps inputs to some output. It is composed of a function symbol (name beginning by a lower case) and a list of terms. Implemented as ANDANTE.LOGIC_CONCEPTS.FUNCTION

```
<Function> ::= [a-z]* (<Term>, ..., <Term>)
```

Predicate: Function that returns either true or false. Implemented as ANDANTE.LOGIC_CONCEPTS.PREDICATE

```
<Predicate> ::= <Function>
```

Constant: Function of arity 0, i.e. number or string. Implemented as ANDANTE.LOGIC_CONCEPTS.CONSTANT

```
<Constant> ::= <Number> | <String>
```

Term: Argument of functions. Implemented as ANDANTE.LOGIC_CONCEPTS.TERM

```
<Term> ::= <Constant> | <Variable> | <Compound term>
```

Atom: Atomic formula as described in first order logic. Implemented as ANDANTE.LOGIC_CONCEPTS.ATOM

```
<Atom> ::= <Predicate> | True | False
```

Compound term: As defined in prolog, compound terms are any terms that are neither unbound variables nor atomic, i.e. constants in our framework. For now, compound terms only refer to functions. Implemented as ANDANTE.LOGIC_CONCEPTS.COMPOUNDTERM.

```
<Compound term> ::= <Function>
```

Literal: Atom, a.k.a. positive literal, or negation of an atom, a.k.a. negative literal.

Clause: Finite set of literals. It is to be understood as the disjunction of all elements of the set. Let A_i, B_i be atoms, the clause can be written as:

$$\begin{aligned} & \{A_1, ..., A_n, \neg B_1, ..., \neg B_m\} \\ \iff \quad & A_1, ..., A_n \leftarrow B_1, ..., B_m \end{aligned}$$

Horn clause: Clause with at most one positive literal. Implemented as ANDANTE.LOGIC_CONCEPTS.CLAUSE

```
<Horn clause> ::= <Definite clause> | <Goal clause>
```

Definite clause: Horn clause with exactly one positive literal.

```
<Definite clause> ::= <Atom> :- <Atom>, ..., <Atom>
```

Goal clause: Horn clause with zero positive literal.

```
<Goal clause> ::= :- <Atom>, ..., <Atom>
```

Goal: The input to our solver. The query for which we need to verify the veracity. Implemented as ANDANTE.LOGIC_CONCEPTS.GOAL

```
<Goal> ::= (<Atom> | <Goal negation>)*
```

Goal negation: The negation of a goal. Implemented as ANDANTE.LOGIC_CONCEPTS.NEGATION

```
<Goal clause> ::= <Atom>, ..., <Atom>
```

Logic Program A collection of Horn clauses. Implemented in ANDANTE/KNOWLEDGE.PY.

```
<Logic program> ::= <Clause>, ..., <Clause>
```

1.2 Inductive Logic Programming concepts

Mode Mode declaration as defined in the progol framework. Mode declarations define language constraints on the new clauses built with Inductive Logic Programming. Implemented as ANDANTE.MODE.MODE.

```
<Mode> ::= <Modeh> | <Modeb>
```

Modeh Mode declaration for the head literal of a clause. Implemented as ANDANTE.MODE.MODEH.

```
<Modeh> ::= modeh ( <Integer>, <MPredicate> )
```

Modeb Mode declaration for body literals of a clause. Implemented as ANDANTE.MODE.MODEB.

```
<Modeb> ::= modeb ( <Integer>, <MPredicate> )
```

MPredicate Predicate whose terms are MTERMS.

MTerm Special Term for Mode declarations. MTerms have TYPE instead of VARIABLE.

```
<MTerm> ::=  <Constant> | <Type> | <MCompoundTerm>
```

MCompoundTerm Compound term whose terms are MTERMS.

Type Object used in the progol framework to tell to what corresponds the Type. + is for input Variable, - is for output Variable and # is for Constant. The string after +-# gives the class which represents the Variable / Constant, e.g. a person or an animal.

```
<Type> ::=  [+-#] <String>
```

1.3 Parser

The terms defined above form together a grammar. The Andante package implements a parser for each term as a single class ANDANTE.PARSER.PARSER. This class relies on the module ANDANTE/GRAMMAR.PY for interpreting input strings into grammatical trees and on the module ANDANTE/GRAMMAR_TREE_VISITOR.PY to transform those grammatical trees into python objects.

It is worth noting that the Andante parser builds upon the PARSIMONIOUS python package [1].

Figure 1 illustrates with an example a trivial use of the parser.

```
In [1]: from andante.parser import Parser
        parser = Parser()
        text = """
        :- begin_bg.
            mortal(X):-man(X).
            man(socrates).
        :- end_bg.
        """
        knowledge = parser.parse(text, rule='background')
        print(knowledge)

        Knowledge object (class: TreeShapedKnowledge)
        Clauses:
           mortal(X) :- man(X).
           man(socrates).
```

Figure 1: Jupyter notebook cell showcasing the use of the Andante parser

2 Logic Programming

Logic Programming is a programming paradigm that allows for computers to reason. The most famous language upholding this paradigm is Prolog.

Section 2.1 describes how one can reason through deduction. Illustrating theory with a simple example.

Section 2.2 presents how one uses the Andante solver, the deduction engine.

2.1 Deduction

Deduction happens when one reasons by drawing deductive inferences, i.e. by applying rules of inference. In Logic Programming, such rules are represented by Horn clauses.

An Horn clause can be noted as:

$$H \leftarrow B_1, \ldots, B_n$$

where H and B_i are atoms. In Logic Programming, Horn clauses are often noted as:

```
H :- B1, ..., Bn
```

The rule is to be read as: if all B_i are true, then H is true. The rule can also be read as: for H to be true, all B_i need to be true. It is worth noting that in the case of a Goal clause, i.e. when there are no H in the clause, the rule becomes: it can never happen that all B_i are true at the same time. Another interesting case is when there exists no B_i. The rule then becomes a fact: H is true.

Of course, in general we use multiple Horn clauses to realize our deduction. This is where comes into action Logic Programs which are nothing more than a collection of Horn clauses.

Also, one may want to deduce the validity of more than a simple atom. Goals allows for just that.

Example

To better understand deductive inferences, let us consider the following Logic Program composed of two Horn clauses:

```
C1 ::= mortal(X) :- man(X).
C2 ::= man(socrates).
```

This Logic Program tells us that (i) all men are mortals and (ii) Socrates is a man.

Now, let us try to deduce the following goal:

```
mortal(X).
```

The question this goal asks is the following: who/what are mortal? From $C1$, we can deduce a new goal: `man(X).`. And from $C2$, we know that the substitution $\{X/socrates\}$ answers the initial query.

2.2 Solver

Once one has defined a logic program, one can query this logic program. The Andante toolbox allows to do so through the ANDANTE.SOLVER.SOLVER class. This class provides two methods to query a logical program. The first method SUCCEEDS_ON tells whether there is a substitution for that query. The second method QUERY gives the substitution for that query.

Figure 2 illustrates both methods in a simple example. This cell is executed in a jupyter notebook after the cell presented in section 1.3. Both PARSER and KNOWLEDGE are defined there.

```
In [2]: from andante.solver import AndanteSolver
        solver = AndanteSolver()
        q = parser.parse("mortal(X).", rule="query")

        # Using method succeeds_on
        success = solver.succeeds_on(q, knowledge)
        if success:
            print("The query succeeded")
        else:
            print("The query failed")

        # Using method query
        print("The solutions for the query are:")
        for solution in solver.query(q, knowledge):
            print('\t', solution)

        The query succeeded
        The solutions for the query are:
                {X: socrates}
```

Figure 2: Jupyter notebook cell showcasing both methods form the Solver class

3 Inductive Logic Programming

Inductive Logic Programming (ILP) is a type of machine learning based on the concept of induction.

Section 3.1 introduces the fundamentals of induction. Section 3.2 gives the progol algorithms that together form the ILP Andante implements. Section 3.3 talks about how induction is conducted in Andante.

3.1 Induction

Induction is defined as the inverse of deduction which is explained in section **??**.

Let us look back the example from section 2.1.

In deduction, from prior knowledge:

Statement A All men are mortal

One deduces a new fact.

Statement B Socrates is mortal

In induction, from some fact:

Statement B Socrates is mortal

One deduces the knew knowledge: Statement A All men are mortal

From this example, one fact is important to notice. Deduction is an sound operation in the sense that if the prior knowledge is correct, the deduced fact is always correct. Induction is not a sound operation. Even if the fact(s) it is based on is (are) true, the induced knowledge may not be correct.

Deduction is making inferences from a general knowledge to a specific instance. Induction is generalizing specific instances into a general knowledge. In the examples provided above, statement A is indeed more general than statement B. To give a more rigorous explanation of the concept of generality, we first need to introduce the notions of substitution and subsumption.

3.1.1 Substitution

Let θ be a substitution of the form $\{v_1/t_1, ..., v_n/t_n\}$. Let F be an arbitrary formula. We note:

$$F' = F\theta$$

where F' is the formula F with each variable v_i of F that is also a variable of θ has been replaced by t_i.

Substitution is implemented in Andante as the class ANDANTE.SUBSTITUTION.SUBSTITUTION. This class handles:

- Unification with its method UNIFY
- Substitution with its method SUBSTITUTE

3.1.2 Clause subsumption

Let C and D be clauses. We say that C subsumes D, that is:

$$C \preceq D \iff \exists\theta, C\theta \subseteq D$$

Using subsumption, we now have a rigorous method to tell whether a clause C is more general than another clause D by testing if C subsumes D.

Taking back the examples from section 3.1, as statement A subsumes statement B, we can say that deduction goes from general to specific and induction goes from specific to general.

3.2 Induction algorithm

This section describes the method used for using induction to learn new knowledge. The learning process of Andante closely follows that of progol. It is composed of three algorithms described in sections 3.2.1, 3.2.2 and 3.2.3. Lastly, section 3.2.4 introduces all metrics related functions used in section 3.2.3.

3.2.1 Cover set alogorithm

The first and main algorithm which glues together the three algorithms is the Cover set algorithm (see 1). It iterates through all positive examples, constructing a theory that covers all positive examples whilst never covering any negative ones. The Andante implementation of this algorithm can be found as the INDUCE method of the ANDANTE.LEARNER.PROGOLLEARNER class.

Algorithm 1: Cover set algorithm

input: h, i, B, M, E

1 If $E = \emptyset$ return B
2 Let e be the first example in E
3 Construct clause $\perp_i$ for e ; `// Algorithm 2`
4 Construct clause H from $\perp_i$; `// Algorithm 3`
5 Let $B = B \cup H$
6 Let $E' = \{e : e \in E \text{ and } B \models e\}$
7 Let $E = E - E'$
8 Goto 1

3.2.2 Algorithm for $\perp_i$

For each new clause added to the theory, it is first necessary to construct a more specific clause called bot_i. This is precisely the role of algorithm 2. This algorithm is implemented in the BUILD_BOTTOM_I method of the ANDANTE.LEARNER.PROGOLLEARNER class.

Algorithm 2: Algorithm to construct $\perp_i$

input: h, i, B, M, E

1 Add $\overline{e}$ to the background knowledge
2 $InTerms = \emptyset$, $\perp_i = \emptyset$
3 Find the first head mode declaration h such that h subsumes a with substitution θ
4 **forall** $v/t \in \theta$ **do**
5 **if** *v corresponds to a #type* **then** replace v in h by t
6 **if** *v corresponds to a +type or -type* **then**
7 replace v in h by v_k
8 where v_k is the variable such that $k = hash(t)$
9 **if** *v corresponds to a +type* **then** add t to the set $InTerms$
10 Add h to $\perp_i$
11 **forall** *body mode declaration b* **do**
12 **forall** *possible substitution θ of variables corresponding to +type by terms from the set InTerms* **do**
13 **repeat**
14 **if** *Progol succeeds on goal b with answer substitution θ'* **then**
15 **foreach** *v/t in θ and θ'* **do**
16 **if** *v corresponds to #type* **then** replace v by b ;
17 **else** replace v in b by v_k where $k = hash(t)$;
18 **if** *v corresponds to -type* **then** add t to the set InTerms ;
19 Add $\overline{b}$ to $\perp_i$
20 **until** *recall times*;
21 Increment the variable depth
22 Goto line 11 if the maximum variable depth has been achieved

3.2.3 Lattice search algorithm

Once the most specific (because of its long boy) clause $\bot_i$ is constructed, we can now compare various possible clauses to add to our previous knowledge. This is the role of algorithm 3. It can be found as the BUILD_HYPOTHESIS method of the ANDANTE.LEARNER.PROGOLLEARNER class.

Algorithm 3: Lattice search algorithm

input: h, i, B, M, E

1 $Open = \{\square\}$, $Closed = \emptyset$

2 $s = best(Open)$, $Open = Open - s$, $Closed = Closed \cup \{s\}$; `// Node selection`

3 **if** $prune(s)$ **then** goto 5 ; `// Node pruning`

4 $Open = (Open \cup \rho(s)) - Closed$; `// Clause refinement`

5 **if** $terminated(Closed, Open)$ **then return** $best(Closed)$; `// End of search`

6 **if** $Open = \emptyset$ **then return** e; `// No generalisation`

7 Goto 2

3.2.4 Metrics for the lattice search algorithm

There can be multiple variations of this algorithm. For instance, depending on whether we have access to negative examples, the notion of best or terminated will be different. In total, there are four functions whose definition may change: best, prune, ρ, terminated.

The Andante toobox implements only the case where we have access to both positive and negative examples. All those functions are implemented in the ANDANTE/HYPOTHESIS_METRICS.PY module.

- Let s be the current state.
- Let C be the clause of state s.
- Let p be the number of positive examples rightly covered by C.
- Let n be the number of negative examples wrongly covered by C.
- Let c be the number of elements in the body of C.
- Let d be a function giving an optimistic estimate of the number of atoms needed to complete the clause.

$$d(C) = \max_{v \in C_{body}} d(v)d(v) = \begin{cases} 0, & \text{if } v\text{is in the head of } C \\ (\max_{u \in U_v} d(u)) + 1, & \text{otherwise} \end{cases}$$

  where U_v are the variables in atoms in the body of C containing v.
- Let h be $d(C)$.
- $g = p - c - h$
- $f = g - n = p - n - c - h$

3.3 Learner

In the Andante package, the ILP learning is done through the ANDANTE.LEARNER.LEARNER class. This class is used through the induce method which produces a the knowledge learned through ILP.

Figure 3 illustrates on a simple example the learning. The cell follows the execution of the cells presented in figures 1 and 2. In particular, PARSER is defined there.

In [3]:
```
text = """
% Mode declarations
modeh(*,daughter(+person,-person)).
modeb(*,parent(+person,-person)).
modeb(*,parent(-person,+person)).
modeb(*,female(+person)).
modeb(*,female(-person)).

% Determinations
determination(daughter/2,parent/2).
determination(daughter/2,female/1).

% Background knowledge
:- begin_bg.
person(ann). person(mary). person(tom). person(eve). person(lucy).
female(ann). female(mary). female(eve). female(lucy).
parent(ann,mary).
parent(ann,tom).
parent(tom,eve).
parent(tom,lucy).
:- end_bg.

% Positive examples
:- begin_in_pos.
daughter(lucy,tom).
daughter(mary,ann).
daughter(eve,tom).
:- end_in_pos.

% Negative examples
:- begin_in_neg.
daughter(tom,ann).
daughter(tom,eve).
:- end_in_neg.
"""
pr = parser.parse(text, rule="andantefile")
H = pr.induce()

print("Knowledge learned")
print(H)
```

```
Knowledge learned
Knowledge object (class: TreeShapedKnowledge)
Clauses:
   daughter(A, B) :- parent(B, A).
```

Figure 3: Jupyter notebook cell showcasing learning on a simple example

4 Interface

Inductive Logic Programming can come as quite cryptic when first glanced at. The documentation available is often not sufficient to understand precisely what is going on. To this purpose, the Andante Interface provides a great way to view in a more intuitive way the learning process. The Andante Interface is implemented in the ANDANTE/INTERFACE.PY module. It relies on the IPYWIDGETS library[1] to provide panels formed from those widgets that the user can interact with.

The Andante Interface comes in 3 panels (figures 4, 5 and 6).

The Andante Interface (see ANDATE.INTERFACE.MAININTERFACE) provides a panel for querying knowledge (see ANDATE.INTERFACE.QUERYINTERFACE). In figure 4, we see that for the query *mother(sylvia, X).*, there are 4 possible substutions.

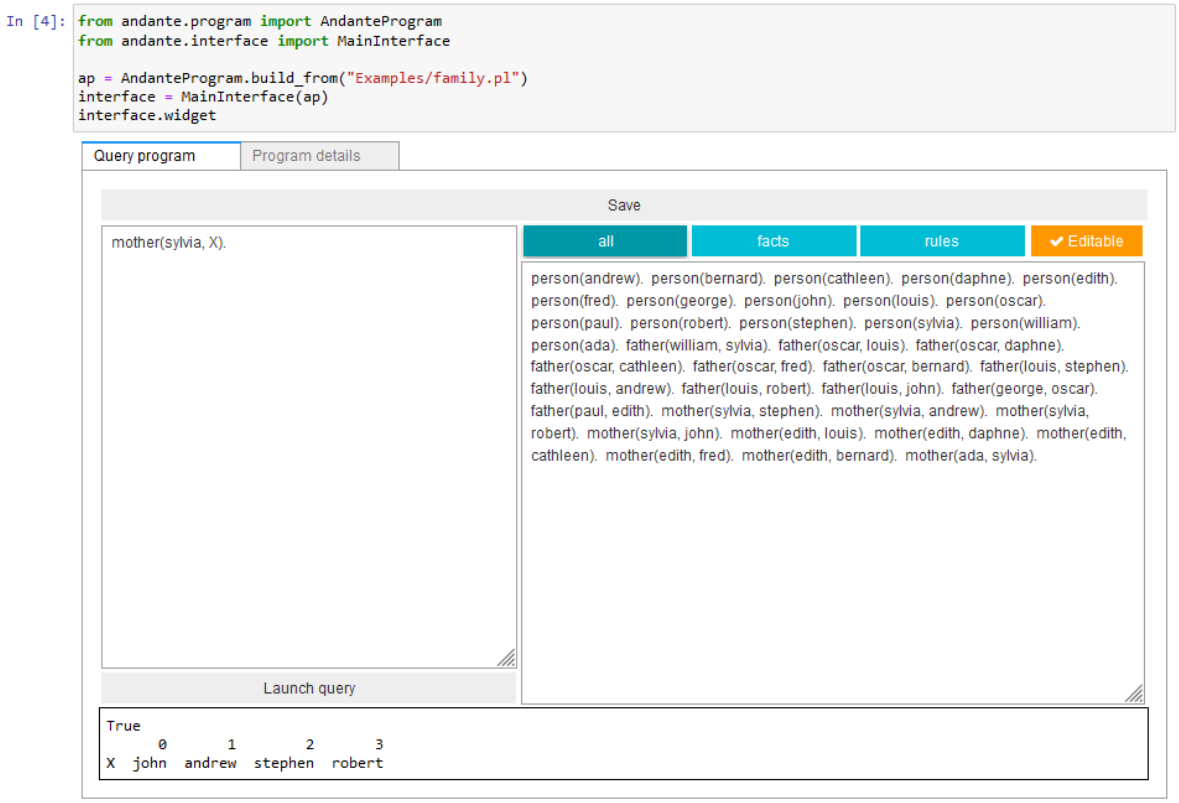

Figure 4: Query panel of the Andante Interface

The second panel (see ANDATE.INTERFACE.DETAILSINTERFACE) shown in figure 5 shows numerous details on the ANDANTEPROGRAM considered. All those details can be edited. The button *Induce* launches the learning process and opens a new panel.

[1]Source: `https://ipywidgets.readthedocs.io/en/stable/`

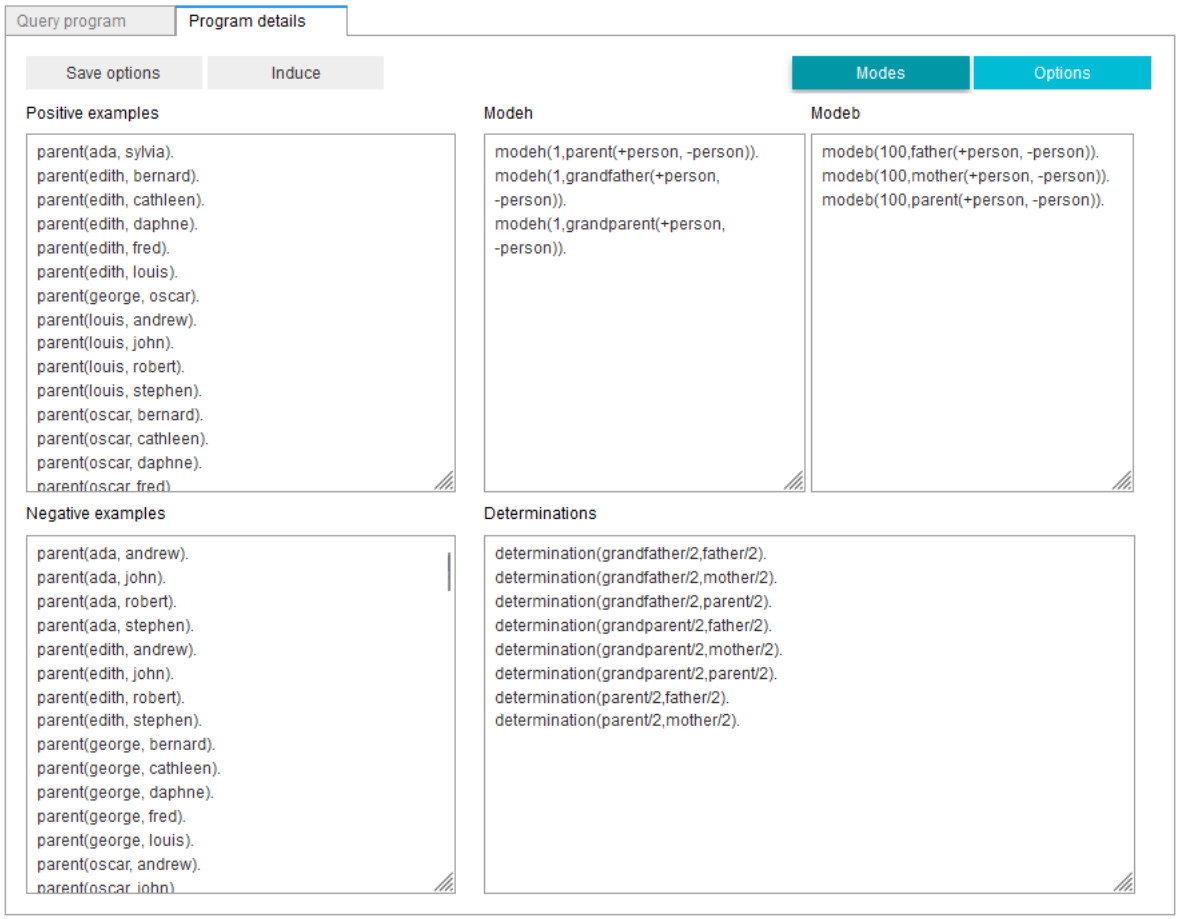

Figure 5: Program details panel of the Andante Interface

The last of the major panels is the *Induction* panel (see ANDATE.INTERFACE.DETAILSINTERFACE), as shown in figure 6. When first opened, this panel offers many details on the learning process that can be played with. In particular, for a particular example, one can find: the bottom clause and all candidate clauses. For a candidate clause, one can find all positive and negatvie examples it covers. In figure 6, one can see for the rule `parent(A, B):- mother(A, B).` all the covered positive examples.

A subpanel (see ANDATE.INTERFACE.QUERYINTERFACE) of this last panel is shown in figure 7. In this panel, one can challenge the knowledge both before and after induction.

5 Package architecture

- **Utilities:** Regroups a few utility classes.

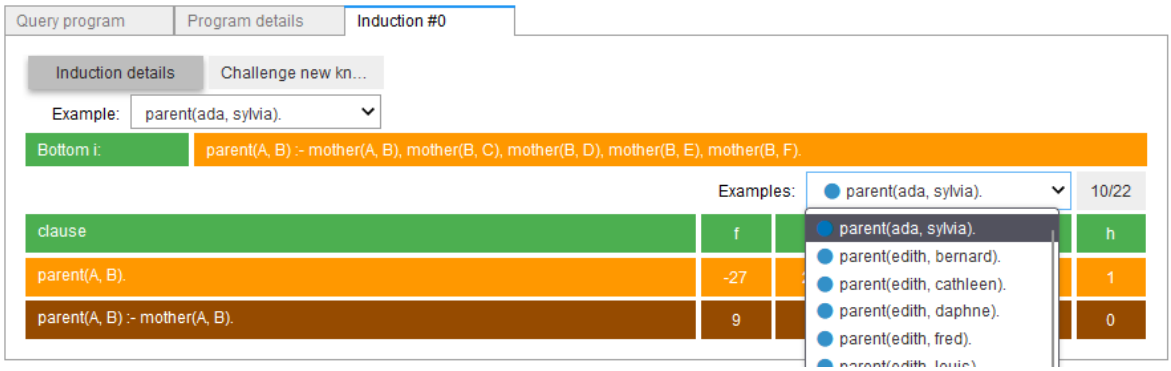

Figure 6: Induce panel of the Andante Interface

Figure 7: Query panel for the induced knowledge of the Andante Interface

  - ANDANTE/COLLECTIONS.PY: Defines ORDEREDSET class.
  - ANDANTE/OPTIONS.PY: Option-related classes.

- **Basic objects:** Regroups all classes implementing terminology as described in section 1.1.

  - andante/logic_concepts.py: Classes for Logic Concept terminology as described in section 1.1.
  - andante/mode.py: Classes for Inducive Logic Concept terminology as described in section 1.2.
  - andante/mathematical_expressions.py: Classes for all mathemitical expressions.
  - andante/knowledge.py: Classes that act as knowledge, i.e. collections of clauses.
- **Parsing:** Regroups modules for parsing text into Python objects.
  - andante/grammar.py: Module for defining the grammar for parsing. The class also allows to transform text into a grammar tree.
  - andante/grammar_tree_visitor.py: Module for turning the grammar tree into Python objects.
  - andante/parser.py: Main module for parsing text. It makes the link between the two previous modules.
- **Solving:** Regroups modules around the engine for deduction.
  - andante/substitution.py: Defines the Substitution class that handles substitution but also unification.
  - andante/solver.py: Implements the engine for deduction.
- **Learning:** Regroups modules around the induction engine.
  - andante/hypothesis_metrics.py: Metrics for choosing the best clause to learn.
  - andante/learner.py: Engine for induction, i.e. learning new clauses.
  - andante/live_log.py: Logs for storing information produced by the induction engine.
- **Interface:**
  - andante/interface.py: Defines interfaces to query, induce, inspect an Andante program.
- **Program:**
  - andante/program.py: Defines the AndanteProgram class that facilitates using the paser, the solver and the learner.

References

[1] Erik Rose. Parsimonious. `https://github.com/erikrose/parsimonious`, 2012.