Imperial College
London

COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Logic Based Learning

Author:
Your Name (CID: your college-id number)

Date: March 18, 2019

1 Inductive Logic Programming

Inductive Logic Programming (ILP) is a research area formed at the intersection of Machine Learning and Logic Programming.

A machine learning problem is a search problem. The task is to compute a solution that would minimise a loss function.

ILP can be seen as a search problem for an hypothesis expressed in logic, that would minimise a loss function.

1.1 Predictive Learning ILP

Predictive ILP can be seen as the usual machine learning binary classification problem

In a **predictive learning task**, we are normally given:

1. A Background Knowledge B
2. A set of examples E that includes positive examples E^+ and negative examples E^-
3. A **hypothesis language** for the search space
4. The notion of **covers** and **reject** relation
5. The notion of quality criterion

The goal is to find a hypothesis, expressed in the given hypothesis language, such that it satisfies all positive examples and none of the negative examples in the dataset.

The notion of **covers relation** depends on the type of learning task. Different ILP paradigms / type include:

1. Learning from entailment
2. Learning from interpretation
3. Learning from answer sets
4. Induction of stable models
5. Brave induction
6. Cautious induction

The covers relation can also be accompanied by a **quality criterion**. This is used when an ILP task is unable to compute hypothesis that cover all given positive examples and none of the negative examples. A quality criterion can be to maximise the test accuracy.

1.2 Concept Learning

Concept learning is a type of logic-based learning that aims to infer a general definition of a concept given examples labelled as member or non-member of the concept.

Sky	Temp	Wind	Water	EnjoySport
Sunny	Warm	Strong	Warm	Yes
Sunny	Warm	Strong	Cold	Yes
Sunny	Cold	Weak	Cold	No
Sunny	Cold	Strong	Warm	Yes
Rainy	Cold	Strong	Cold	No

In the example above, the concept that we want to learn is *What are the days my friend enjoys water sports?*

Those that have *yes* under the category *Enjoy Sport*, are positive examples of the concept that we want to learn and the rest are negative examples.

The **covers relation** $c(h, E)$ is the set of examples (rows in the tables) that are covered by the hypothesis h.

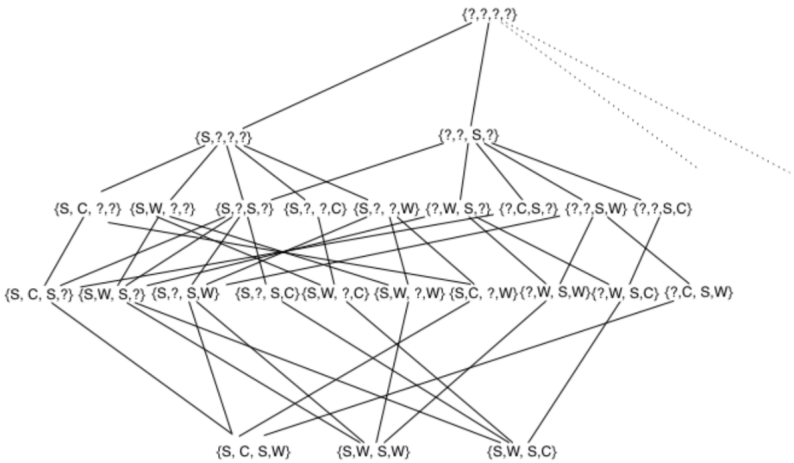
$h_1 = EnjoySport \leftarrow Sky = Sunny$ $\quad c(h_1, E) =$ rows 1 2 3 4

$h_2 = EnjoySport \leftarrow Sky = Sunny \wedge Wind = Strong$ $\quad c(h_2, E) =$ rows 1 2 4

$h_3 = EnjoySport \leftarrow Sky = Sunny \wedge Temp = Warm \wedge Wind = Strong$ $\quad c(h_2, E) =$ rows 1 2

1.3 Version Space

Using the example in section 1.2, we can construct a **version space model**.



In the diagram above, the four values {?, ?, ?, ?} correspond to the four attributes Sky, Temp, Wind and Water.

For examples, $\{S, ?, ?, ?\}$ correspond to the rule $EnjoySport \leftarrow Sky = Sunny$.

Going down the version space the parent node becomes more specific. For example, $\{S, W, ?, ?\}$ is more specific than $\{S, ?, ?, ?\}$. $\{S, W, ?, ?\}$ is called a specialisation of $\{S, ?, ?, ?\}$.

Specialisation is able to eliminate the coverage of a negative example. However, hypothesis should not be specialised too much as it risks to eliminate also the coverage of true positive examples.

The aim of concept learning is to navigate through the **version space** in the search for the best hypothesis.

1.4 ILP Search

To define an ILP learner as a search of program clauses we can make use of 3 main components:

1. Structure of the hypothesis space
2. Search Strategy
3. Heuristics

Given the property of the structure, search strategies can be defined. Search strategies can be uninformed strategies (depth-first search, random walk) or heuristic search.

The heuristic aspect of an ILP learner is what decides when to stop. In the case of learning with no noise in the examples, such stopping criteria is when an hypothesis is found that covers all the positive examples and none of the negative examples.

In the context of **definite clauses**, the structure of hypothesis space is characterised by a **generality relation**.

1.4.1 Generality Relation

The concept of generality is important to any search algorithm over version spaces

An hypothesis h is said to be more general than another hypothesis h' iff all (positive) examples covered by h' are already covered by h. We can write $h \geq h'$ to express h is more general than h'.

The generality relation over hypothesis gives the version space the shape of a lattice. Hypothesis at the bottom covers just the given positive examples. The empty hypothesis at the root covers all the examples.

We can search for the best hypothesis without constructing the full version space, using the underlying principle of generality:

1. If h covers a negative examples, then any $g \geq h$ also cover the negative example
2. If h does not cover some positive examples, then any $s \leq h$ does not cover the

positive examples

1.4.2 Search Strategy

The algorithm used is called **candidate elimination algorithm** that defines how to prune the version space by looking at coverage of current hypothesis with respect to a given set of examples.

The algorithm would start from:

1. A set G including at the beginning all the possible hypothesis
2. A set S including at the beginning no hypothesis

For each positive example in $e \in E^+$:

1. Find hypothesis in S that does not cover e and apply one step generalisation
2. Remove hypothesis in G that does not cover e

For each negative example in $e \in E^-$:

1. Find hypothesis in G that cover e and apply one step specialisation
2. Remove hypothesis in S that covers e

1.5 ILP Search First Order Logic

1.5.1 Generality Relation

We can define a different generality relation for first-order theories. The generality relation between theories is defined in terms of **entailment**.

A clause C is more general than another clause D if $C \models D$.

Two principles for searching hypotheses also applies to the entailment relation:

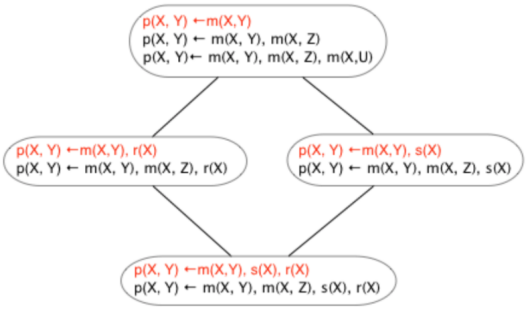
1. If an hypothesis h is **consistent**, does not cover any negative examples, then any specialisation of h is also consistent

2. If an hypothesis h is **complete**, covers all positive examples, then any generalisation of h is also complete

Logical entailment is in general undecidable. Therefore we need to use the notion of **subsumption** between clauses to compute entailment relation.

A clause C subsumes D if there exists a θ such that $C\theta \subseteq D$

It has been shown that the subsumption relation over definite clauses defines a lattice structure.



Each node contains multiple clauses that are equivalent. For example

$$p(X,Y) \leftarrow m(X,Y) \equiv p(X,Y) \leftarrow m(X,Y),\ m(X,Z)$$

by substituting Z with Y

Moving from a node to its parent node is a generalisation step.

Moving from a node to one of its children nodes is a specialisation step.

1.5.2 Search Strategy

Traversing the lattice can be done in two ways:

1. Bottom Up strategy: Applying a generalisation operator
2. Top Down strategy: Applying a specialisation operator

Consequently, ILP learners for definite clauses have been classified into top-down and bottom-up learners.

Specialisation Operator – Shapiro ρ

This operator essentially refines or specialises a clause by **adding a literal to the body** of the current clause or **applying a substitution**.

For example, the concept to learn is $daughter(X,\ Y)$. Applying the specialisation operator generates various possible specialisations.

$daughter(X,\ X)$	Applying a substitution $\{Y/X\}$
$daughter(X,\ Y) \leftarrow parent(X,\ Y)$	Adding a new body literal

Generalisation Operator – Plotking's least general generalisation (lgg)

lgg takes two clauses and compute a third clause that is the most specific of all the clauses that are generalisations of the given initial two clauses.

We can define the lgg between two predicates only if the predicates are the same. For example, given two sets

$$\{f(t,a),\ \neg p(t,a),\ \neg m(t),\ \neg f(a)\} \qquad \{f(j,p),\ \neg p(j,p),\ \neg m(j),\ \neg m(p)\}$$

The lgg is defined as

$$\{f(X,Y),\ \neg p(X,Y),\ \neg m(X),\ \neg m(Z)\}$$

where

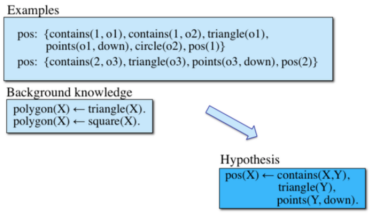
$$lgg(t,j) = X \qquad lgg(a,p) = Y \qquad lgg(t,p) = Z$$

Generalisation Operator – Inverse Resolution

The idea of inverse resolution is to compute clauses from given clauses that reflect an inverse operation of the standard resolution rules

1.6 Learning from Interpretation

In the paradigm of learning from interpretations, examples are assumed to be full interpretations. All the information relevant to an example is included in the example itself. Any ground atom not included in an interpretation is assumed to be false.

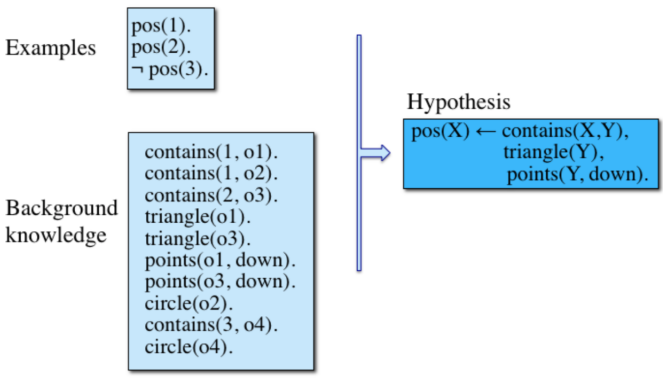


A hypothesis (or solution) is a clause theory that covers all the given positive examples.

Coverage in this case mean that each positive example is a model of the hypothesis.

1.7 Learning from Entailment

In the paradigm of learning from entailment, examples are single ground facts and background knowledge can be related to the given examples.



The task is to find an hypothesis that, together with the background knowledge entails each positive example and does not entail any of the negative examples.

$$B \cup H \models e^+ \qquad \forall e \in E^+$$
$$B \cup H \not\models e^- \qquad \forall e \in E^-$$

2 Hybrid Abductive Inductive Learning (HAIL)

In the context of **concept learning**, various rule-based algorithms have been proposed. This algorithms learn a concept in terms of rules that can be extracted using a **version space** and a **notion of coverage** of the observed data.

There are **Top-Down** learners using the notion of θ**-subsumption** such as FOIL and HYPER.

There are **Bottom-Up** learners using the notion of **lgg** or **inverse resolution**.

2.1 PROGOL

PROGOL uses a different approach that combines **top-down** and **bottom-up** mechanisms by using the notion of **inverse entailment** and **general-to-specific search** through a refinement lattice of θ**-subsumption relation**.

The focus of PROGOL is to support the learning of concepts whose instances are directly observed, known as **observation predicate learning**.

2.1.1 Inverse Entailment

The idea of inverse entailment comes from the observation that for clauses B, H, E

$$B \wedge H \models E \equiv B \wedge \neg E \models \neg H$$

For a positive example e^+, we can compute a set of ground literals entailed by $B \wedge \neg e^+$

$$B \wedge \neg e^+ \models \neg Bot(B, e^+)$$

This is called the **negation of Bottom Set**. The **Bottom Set** is generated by negating every literal in $\neg Bot(B, e^+)$

We can refer to Bottom Set as the ground clause $ground(h_\perp)$, then

$$B \wedge \neg e^+ \models \neg ground(h_\perp) \Rightarrow B \wedge ground(h_\perp) \models e^+$$

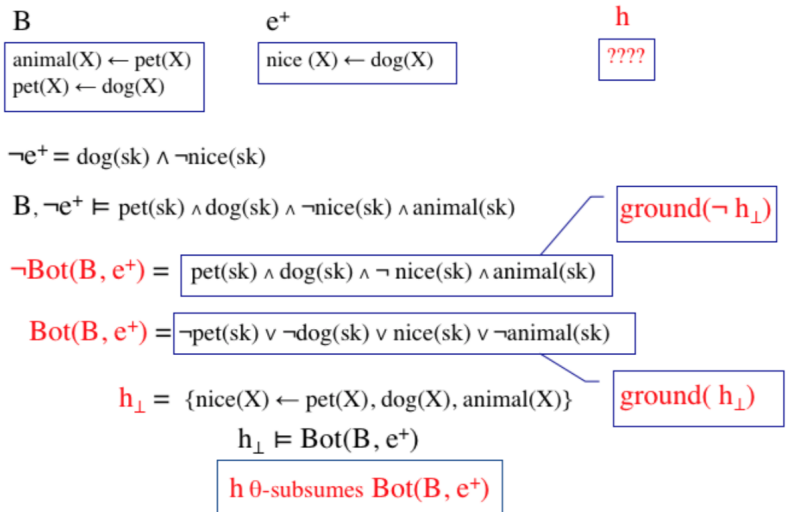
If we replace every constant in $ground(h_\perp)$ with a unique variable, we obtain a universally quantified clause $h_\perp$ that θ-subsumes the Bottom Set. Hence, $h_\perp$ correspond to the **most specific unground hypothesis** at the bottom of the lattice space that

θ-subsume $Bot(B, e^+)$.

Once $h_\perp$ has been generated, the next step is to compute more general clause that subsume this bottom set through top-down refinement. A hypothesis H is **derivable** by bottom generalisation from B and e^+ if and only if $H \succeq Bot(B, e^+)$

Calculating the Bottom Set allows for a more efficient search for hypotheses because it limits the search in the sub-lattice space delimited by the Bottom Set as bottom element and the empty clause as the top element.

Example



The task aims to learn the concept of *nice*. The task here is an Observation Predicate Learning (OPL) as it aims to learn the definition of a predicate that is directly observed.

From the above example, $h_\perp = nice(X) \leftarrow pet(X), dog(X), animal(X)$. Any clause that subsumes now this bottom hypothesis is a possible solution for this particular learning task. For instance, $nice(X) \leftarrow animal(X)$ is a **derivable** hypothesis because it subsumes $h_\perp$

2.1.2 Language Bias

Language Bias is a specification of the language in which the hypothesis is expected to be written. It is used to limit the search space to those hypotheses that are of genuine interest.

Progol allows for the specification of such a bias in terms of what are known as **mode declarations**. For example

$$modeh(1, grandfather(+p, +p))$$
$$modeb(1, father(+p, -p))$$
$$modeb(1, parent(+p, +p))$$

A mode declaration is a template for a literal that can appear in an hypothesis, it specifies:

1. the predicate e.g. $grandfather$
2. the type of arguments of the predicate e.g. p
3. the mode of the terms e.g. $+(input)$, $-(output)$, $\#(constant)$

$modeh$ states that the rule to learn will have to have $grandfather$ as head predicate. $grandfather(+p,+p)$ means that it takes two input variables of type person (p)

The two mode bodies $modeb$ state that predicates $father$ and $parent$ can appear in the body of a learned rule.

$parent(+p,+p)$ takes two input variables, but it appears in the body so the input variable can either:

1. Link to an output variable of a predicate that appear before it in the body of the rule
2. Link to an input variable in the head.

2.2 Non-Observable Predicate Learning

In Observable Predicate Learning, the head predicate of the learned rules is the same as the predicate of the observed examples. The task in section 2.1.1 is an example of OPL, the hypothesis to compute ($nice$) is defined using the same predicate used in the example $nice(X) \leftarrow dog(X)$

In Non-Observable Predicate Learning, the hypothesis to compute uses different predicates as that used to express the examples. We would like to learn about concepts that are different from the observed predicate.

2.2.1 NOPL and Inverse Entailment

Consider using inverse entailment in the following example

B	e^+
hasbeak(X) ← bird (X) bird(X) ← vulture (X)	hasbeak(tweety)

The Bottom Set is generated by finding a set of clauses entailed by $B \wedge \neg hasbeak(tweety)$.

If we use full first-order logic resolution, we would be able to compute the full set of consequences.

$$B \wedge \neg hasbeak(tweety) \models \neg hasbeak(tweety) \wedge \neg bird(tweety) \wedge \neg vulture(tweety)$$

However if we use SLD of PROLOG, it would only derive positive literals and therefore the computed Bottom Set is

$$B \wedge \neg hasbeak(tweety) \models \neg hasbeak(tweety)$$

Solution

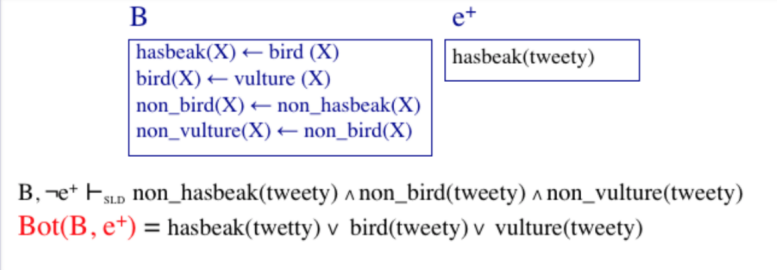
To solve the problem with PROGOL, we can extend the background knowledge B with equivalent contrapositive clauses for each rule. For instance,

$$p \leftarrow q \wedge r \equiv \neg q \leftarrow \neg p \wedge r \equiv \neg r \leftarrow \neg p \wedge q$$

We can then encode this it PROLOG using new names

$$p \leftarrow q \wedge r \equiv non\_q \leftarrow non\_p \wedge r \equiv non\_r \leftarrow non\_p \wedge q$$

For the previous example, we can extend B to



Then PROGOL will be able to generate the correct Bottom Set

2.3 PROGOL 5

We can now briefly consider the general idea of the algorithm that is behind PROGOL5.

Since the assumption is that programs are definite clauses (and therefore monotonic), the algorithm uses a coverage loop approach. In each iteration:

1. A positive examples e^+ known as the **seed example** is chosen from E^+.

2. A hypothesis H is calculated to cover the seed example.

3. H is added to the background knowledge B. All other positive examples in E^+ that is covered by $B \cup H$ will be removed.

4. Another e^+ is chosen for E^+ for the next iteration.

The process is repeated until E^+ becomes empty.

The computation of hypothesis from a seed example in step 2, can be summarised in 3 steps:

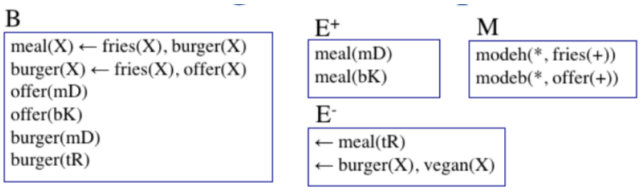
1. The **Start Set** extend the background knowledge B with equivalent contrapositive clauses for each rule. Then it performs a SLD query for each mode head in its negated form. Each successful ground substitution gives the head of a hypothesis $a = s\theta$.

2. The **Bottom Set** uses the substitution from **Start Set**, θ, to derive ground body

predicates that are compatible with the mode body b_1, b_2.... The output combines **Start Set** and **Bottom Set** to form a bottom clause $a \leftarrow b_1, b_2, ..., b_n$

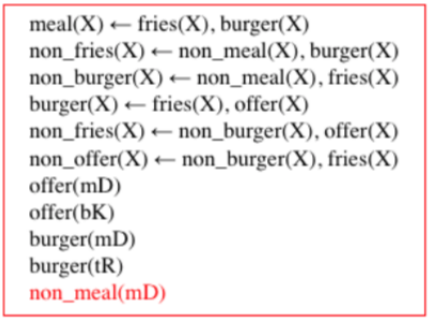
3. **Search** is where the negative examples are taken into consideration together with the positive examples. It replaces the constants in the **learned bottom clause** with variables. Then search for a hypothesis H such that H subsumes the **ground bottom clause**, and $B \cup E^+ \cup E^- \cup H$ is consistent.

Example

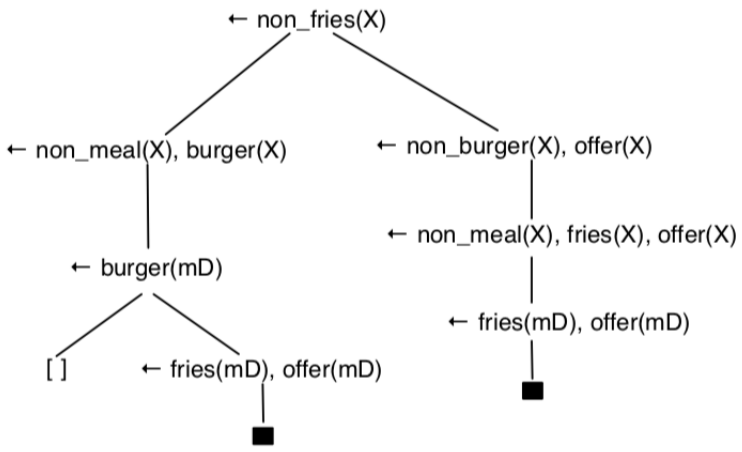


The first example $meal(mD)$ is chosen as the **seed example**

Start Set: The background knowledge is augmented with the contrapositive rules and the negated example is added

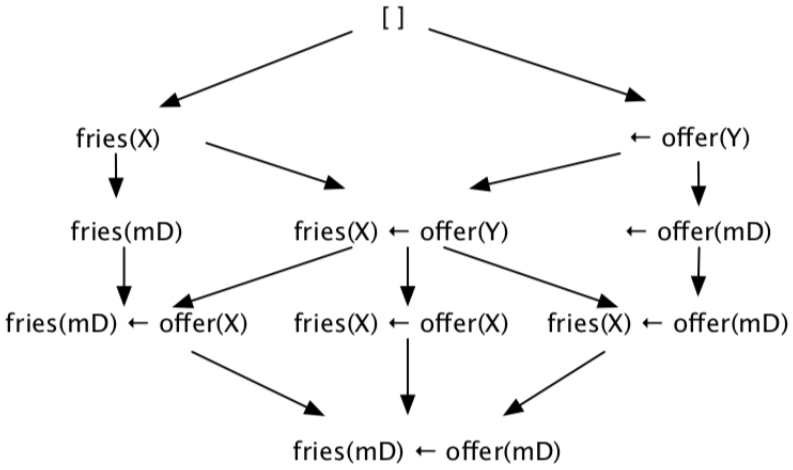


The mode head declaration has the predicate $fries(X)$, and therefore a SLD query in its negated form is performed, $\leftarrow non\_fries(X)$. This returns a unification $\{X/mD\}$, and so $fries(mD)$ is a possible head for the hypothesis.



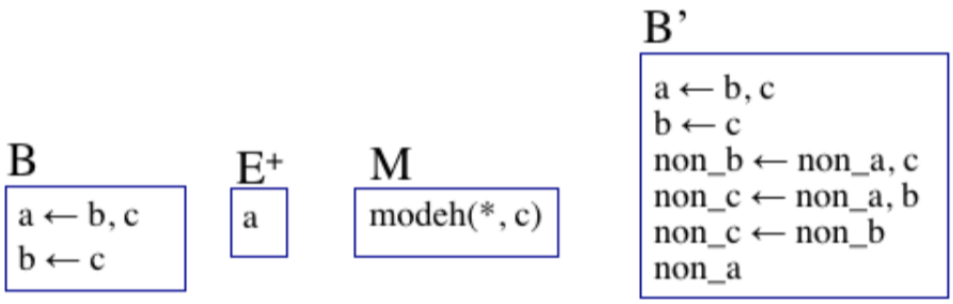
Bottom Set: Given this unification, the program derives ground body predicates that are compatible with the mode body $offer(X)$. It runs the query $\leftarrow offer(mD)$ and succeeds. Therefore the Bottom Set $fries(mD) \leftarrow offer(mD)$ is generated.

Search: It replaces the constants in the learned bottom clause with variables, $fries(X) \leftarrow offer(X)$. Then search through the lattice space for the hypothesis.

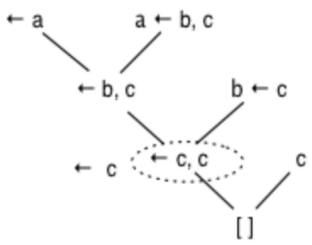


2.4 Incompleteness of Start Set

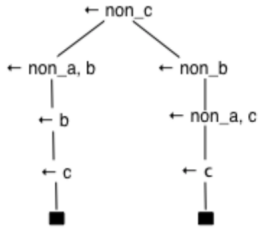
The use of contrapositive mechanism in StartSet is only applicable and works in limited cases. Specifically, it only works when the head predicate needs to be used only once in the deductive proof.



It is obvious that $B \wedge \neg a \models \neg c$. To proof this, we can show$B \wedge \neg a \wedge \neg c \models \bot$ using resolution. Notice though that in the resolution derivation the atom c needs to be used **twice**.



However if we apply SLD derivation, it fails to proof $B \wedge \neg a \models \neg c$.



For this reason, PROGOL5 is incomplete.

2.5 HAIL

A much more powerful learning approach can be achieved by integrating abductive and inductive inference. This is **Hybrid Abductive Inductive Learning (HAIL)**.

The underlying idea is to replace the Start Set procedure with a full abductive reasoning procedure. Given a background knowledge and a seed example, the **heads of a Kernel Set** hypothesis can be computed directly using abduction instead of contrapositives, with the seed example as observation to explain by abductive derivation.

The **body literals of the hypothesis** can still be generated as in PROGOL5

The two sets together form a **Kernel Set**. Notice that the Kernel Set is ground.

HAIL has a similar structure to PROGOL5. The algorithm uses a coverage loop approach, in each iteration:

1. A positive examples e^+ known as the **seed example** is chosen from E^+.

2. An abductive proof procedure that takes B and a set of abducibles to produce possible explanations for e^+, which is a set of positive ground abducibles.

3. B and $\neg e^+$ are used to compute ground instances of the mode body predicates according to the language bias. This will generate a set of ground clauses known as the **Kernel Set** $Kernel(B, e^+)$.

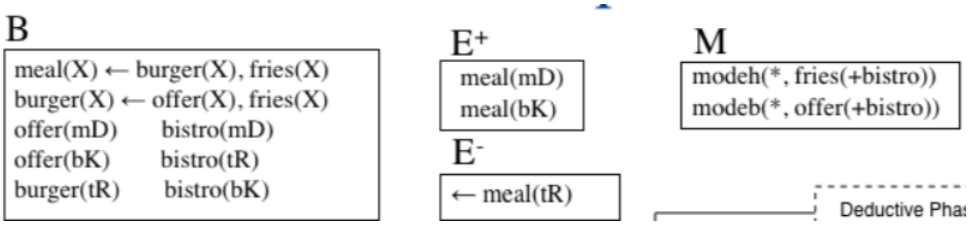
4. The unground version of the Kernel Set forms the bottom element of the lattice. Then the search will find a hypothesis H that subsumes the Kernel Set.

5. H is added to the background knowledge B. All other positive examples in E^+ that is covered by $B \cup H$ will be removed.

6. Another e^+ is chosen for E^+ for the next iteration.

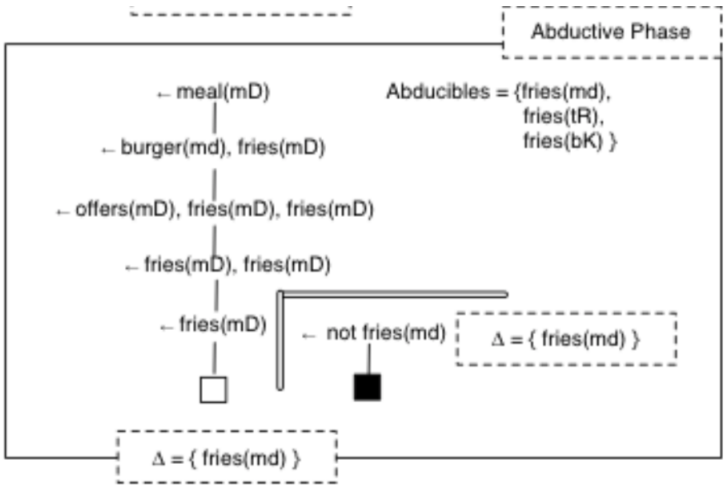
The process is repeated until E^+ becomes empty.

Example



The first example $meal(mD)$ is chosen as the **seed example**.

We then run an abductive proof procedure by setting $meal(mD)$ as **goal**, using B as **background knowledge**, and using ground instances of mode head predicates as **abducibles**



The next step of HAIL is the deductive phase. Considering the unification in the abductive step ($\{X/mD\}$), and prove if instances (with this unification) of mode body literals ($offer(X)$) can be derived by $B \wedge \neg e$ ($B \wedge \neg meal(mD)$).

In this case, we have to show $\leftarrow offer(mD)$ can be derived by $B \wedge \neg meal(mD)$ which is obviously true.

A ground **Kernel Set** is computed, by combining the head predicate in the first step and the body predicates in the second step.$Kernel(B,e) = fries(mD) \leftarrow offer(mD)$.

The final step tries to compute a more general hypothesis, within the same language bias, that θ-subsumes this ground Kernel set. The search now starts from the top and traverse the lattice downwards whenever the current clause is not good enough. (it covers some negative examples)

2.6 Using Abduction in the Induction Step

Once the Kernel set is computed a solution is an set of clauses that θ-subsumes the Kernel Set theory. The computation of the most compressed hypothesis can also be performed using an abductive proof procedure. The procedure is as follow:

1. A Theory T is constructed from the Kernel Set. For example a Kernel set may have

several clauses

$$h_1 \leftarrow b_{11},\ b_{12},\ ...,\ b_{1n}$$
$$h_2 \leftarrow b_{21},\ b_{22},\ ...,\ b_{2n}$$
$$h_3 \leftarrow b_{31},\ b_{32},\ ...,\ b_{3n}$$

1.1 For each clause in Kernel Set. Use the same head predicate h, then for each body literal b_{ij}, add $try(i,\ j,\ X^{ij}$. Where X^{ij} are the variables included in b_{ij}. Then include $use(i,0)$ in the body.

For the first clause in the example, T will include

$$h_1 \leftarrow use(i,0),\ try(1,1,X^{11}),...,\ try(1,n,X^{1n})$$

1.2 For each $try(i,j,X)$, add two clauses

$$try(i,j,X) \leftarrow not\_use(i,j) \qquad and \qquad try(i,j,X) \leftarrow use(i,j)$$

2. In the abduction proof, use is an abducible predicate where $use(i,j)$ means j body literal in clause i is needed.

Example

Using the Kernel Set from example 2.5, $fries(mD) \leftarrow offer(mD)$ can be change to unground $fries(X) \leftarrow offer(X)$.

The set of Theory T generated from this clause is

$$fries(X) \leftarrow use(i,0) \wedge try(1,1,X)$$
$$try(1,1,X) \leftarrow use(1,1)$$
$$try(1,1,X) \leftarrow not\_use(1,1)$$

The abduction proof is then run using $B \cup T$, where use predicates are the abducibles.

Instances of the form $use(i,0)$ in the abductive answer indicate that the clause i in the Kernel Set is needed.

Instances of the form $use(i,j)$ indicate that the body literal j in the clause i of the kernel Set is needed.

In the example if we run an abductive inference on the given abductive task with E as goal, we would get as answer the abducibles $\{use(1,0),\ use(1,1)\}$

3 Top-directed Abductive Learning

We now focus on a more recent class of logic-based learning, which can learn normal logic programs (with negation as failure) and not just definite logic programs, called

Meta Level Learning.

Meta Level Learning is characterised by the two features:

1. Mode Declarations are represented by **top theories** T

2. Learning task is computed by as abductive search on background knowledge B extended with top theories T

There are different approaches to meta level learning, one of which is the **TopLog** system.

3.1 TopLog

In TopLog, the language bias is expressed as a logic program, called **top theory**, instead of a set of meta-logical facts.

The computation of a hypothesis is given by an inference process over the top theory together with the given background knowledge.

Consider the following database B and meta-logical mode declaration M,

$$M = \begin{cases} modeh(penguin(+any)) \\ modem(can(+any, \#ability)) \end{cases} \qquad B = \begin{cases} B1 : can(a, fly) \\ B2 : can(b, swim) \\ B3 : ability(fly) \\ B4 : ability(swim) \end{cases}$$

To convert the set of metal-logical facts to top theory consist of two steps:

1. Each $modeh$ is turned into a head with a non-terminal $\$body()$ predicate

2. Each $modem$ defines $\$body()$ in terms of the mode body declarations given in M.

So for instance the above mode declaration M will become Top Theory

$$T = \begin{cases} T1 : penguin(X) \leftarrow \$body(X) \\ T2 : \$body(X) \leftarrow \\ T3 : \$body(X) \leftarrow can(X, A), ability(A), \$body(X) \end{cases}$$

Given an example $e^+ : penguin(b)$, the algorithm constructs a SLD derivation of e^+ from $B \cup T$. This will construct two possible derivations using [T1, T2] and [T1, T2, T3, B2, B4], and from this derivations, we can construct hypothesis

$$H_1 : penguin(X)$$
$$H_2 : penguin(X) \leftarrow can(X, swim)$$

Note that the type predicate are omitted at the end as they are not explicitly included in the given bias

3.1.1 Limitations of TopLog

Expressing language bias as logic programs is more expressive. This gives TopLog the ability to overcome limitations of PROGOL5.

But on the other hand, the approach has its own limitations:

1. It is mainly tailored for observation predicate learning

2. It cannot be applied to normal logic programs or learning tasks whose background knowledge is expressed a normal logic program.

3.2 Nonmonotonicity

Learning algorithms such as PROGOL5 and HAIL uses a sequential covering loop approach. They learn a set of clauses by considering a single seed example at the time and adding a new clause in a subsequent iteration. The core assumption with these learning algorithms is **monotonicity**.

Monotonicty means given a program composed of a set of definite clauses, the addition of new definite clauses to it preserves the set of consequences of the initial program.

Using the **negation as failure** operator will fail the monotonicity assumption. In this case all learning approaches based on sequential covering loop would not be appropriate.

3.3 Top-directed Abductive Learning

A top-directed abductive learning approach has been proposed in order to overcome the limitations of TopLog. It is one of the first non-monotonic learning approaches capable of learning normal logic programs.

3.3.1 Representing modes as Data Structure

To generate abductive top theories, we have to define a translation function that maps **rules** (compatible with given mode declaration) to **data structure** (use as abducibles).

The data structure is a tuple composed of four elements:

1. Identifier of specific mode

2. List of constants in the predicate of specific mode

3. List of input variables in the predicate of specific mode

4. List of output variables in the predicate of specific mode

Consider the rule $p(X) \leftarrow q(X,Y),\ r(Y,a)$, and the set of mode declaration

$$M = \begin{cases} m1 : modeh(p(+any)) \\ m2 : modeb(r(+any,\ \#any)) \\ m3 : modeb(q(+any,\ -any)) \end{cases}$$

It can be translated to the data structure

$$< m1,\ [],\ [X],\ [] >,\ < m3,\ [],\ [X],\ [Y] >,\ < m2,\ [a],\ [Y],\ [] >$$

It is possible to simplify the data structure representation by omitting the list of output variables. We can assume all distinct variables in a rule are numbered from left to right, and keep track only of which variable in the rule an input variable argument of a literal refer to.

Consider the rule $uncle(X,Y) \leftarrow father(Z,Y),\ sibling(Z,X),\ gender(X,m)$, and the set of mode declaration

$$M = \begin{cases} m1 : modeh(uncle(+person,\ +person)) \\ m2 : modeb(father(-person,\ +person)) \\ m3 : modeb(gender(+person,\ \#mf)) \\ m4 : modeb(sibling(+person,\ +person)) \end{cases}$$

We first number distinct variable in the rule from left to right

$$uncle(X_1,X_2) \leftarrow father(X_3,X_2),\ sibling(X_3,X_1),\ gender(X_1,m)$$

Omitting the list of outputs, the data structure now becomes

$$< m1,\ [],\ [] >,\ < m2,\ [],\ [2] >,\ < m4,\ [],\ [1,\ 3] >,\ < m3,\ [m],\ [1] >$$

There are two constrains for this simplification:

1. Output variables cannot appear as input variables in the same predicate

2. Input variable in a body predicate is either linked to input variable in head predicate or to an output variable of another body predicate before it

3.3.2 Abductive Top Theory

We can now represent a clause, compatible with a mode declaration, into a list of tuples. We can then form an abducible using this list of tuples with the predicate $rule()$.

The generation of abductive top theory take a set of mode declaration as input.

Each $modeh$ is turned into a head with a $body$ predicate that has input i) list of input of modeh ii) tuple encoding of modeh

The $body$ predicate is defined in terms of all possible mode body declarations that

are given in the language bias.

For each $modeb$, there is a rule that defines $body$ to be true provided that appropriate instantiation of $modeb$ can be proved. If this is true, the structured data for this body literal is added to the $RuleSoFar$ term.

Example

Consider the mode declaration

$$M = \begin{cases} m1 : modeh(*, p(+any)) \\ m2 : modeb(*, q(+any,\ \#any)) \\ m3 : modeb(*, q(+any,\ -any)) \end{cases}$$

The top theory generated is

$$T = \begin{cases} T1 : p(V1) \leftarrow body([V1],\ [(m1,\ [],\ [])] \\ T2 : body(InputSoFar,\ RuleSoFar) \leftarrow rule(RuleSoFar) \\ T3 : body(InputSoFar,\ RuleSoFar) \leftarrow q(V1,\ V2), \\ \qquad\qquad\qquad\qquad\qquad\qquad link([V1],\ InputSoFar,\ Links), \\ \qquad\qquad\qquad\qquad\qquad\qquad append(RuleSoFar,\ (m3,\ [],\ Links,)\ NewRule), \\ \qquad\qquad\qquad\qquad\qquad\qquad append(InputSoFar,\ [V2],\ NewInputs), \\ \qquad\qquad\qquad\qquad\qquad\qquad body(NewInputs,\ NewRule) \\ T4 : body(InputSoFar,\ RuleSoFar) \leftarrow q(V2,\ A), \\ \qquad\qquad\qquad\qquad\qquad\qquad link([V2],\ InputSoFar,\ Links), \\ \qquad\qquad\qquad\qquad\qquad\qquad append(RuleSoFar,\ (m2,\ [A],\ Links,)\ NewRule), \\ \qquad\qquad\qquad\qquad\qquad\qquad append(InputSoFar,\ [],\ NewInputs), \\ \qquad\qquad\qquad\qquad\qquad\qquad body(NewInputs,\ NewRule) \end{cases}$$

3.3.3 Top-directed Abductive Learning

The basic algorithm of top-directed abductive learning consists of three steps.

The first step generates an abductive top-theory T from given mode declarations M.

The second step performs a pure abductive inference. The **background knowledge** for abduction is $B \cup T$. The set of **abducibles** is given by all the possible ground instances of the atom $rule()$. The set of **integrity constraints** is the same set that might appear already on the given initial learning task. The **goal** is the entire set of examples (negative examples are written as negated literals). All the examples are considered in one go because being the learning non-monotonic.

The third step consists of applying the inverse transformation to the abductive answer and generate the corresponding hypothesis.

4 Logic Based Learning is ASP

4.1 Recap

Grounding

We can generate a set of ground instances of rules in P incrementally.

In each step, we only generate rules R if each atom a in $body^+(R)$ is already the head of another ground rule.

Safety

A rule R is safe, if every variable in R occurs in at least one atom in $body^+(R)$.

Reduct

The reduct of any ground normal logic program P with respect to a set of atoms X is constructed in two steps:

1. Remove any rule in P whose body contains the negation of an atom in X
2. Remove any negation from remaining rules in P

Stable Model

An interpretation X is a stable model of P if and only if X is the unique least Herbrand Model of $ground(P)^X$

Brave and Cautious Entailment

An atom A is bravely entailed by P if it is true in at least one stable model of P.

An atom A is cautiously entailed by P if it is true in every stable model of P

4.2 Answer Set Programming

The stable model semantics grew into the field that is now known as Answer Set Programming (ASP).

ASP allow an extended syntax in which constraints, aggregates, classical negation and optimisation statements are allowed

4.2.1 Constraints

Constraints can be used as a way of ruling out Answer Sets which are not intended solutions of the problem we are trying to represent.

$$:-b_1,\ b_2,...,\ b_m,\ not\ c_1,\ not\ c_2,...,\ not\ c_n$$

When computing the reduct, the head of the constraint is replaced by $\bot$

4.2.2 Choice Rules

Aggregates are a construct in ASP which allow the mapping of a set of atoms contained in an Answer Set to an integer.

We only consider the most common aggregate **count** and restrict it to only appearin the head rather than the body.

The aggregate $a\{h_1,\ h_2,...,\ h_n\}b$ is satisfied by interpretation X if $a \leq |\{h_1,\ h_2,...,\ h_n\} \cap X| \leq b$

When computing the reduct of rule R with aggregate A as its head:

1. If A is not satisfied with respect to the interpretation X, we remove the rule R.

2. If A is satisfied with respect to X, then we replace R with one rule for each atom in A which is true in X

4.2.3 Optimisation Statements

Optimisation statements are used to specify which Answer Sets of the program are preferred over others.

Optimisation statements are of the form

$$\#minimize[a_1 = w_1,...,\ a_n = w_n]$$
$$\#maximize[a_1 = w_1,...,\ a_n = w_n]$$

Where a are ground atoms and w is the associate weight.

The solver will then search for an optimal answer set that minimise or maximise the weighted sum of the atoms.

Note that weak constraints are equivalent to optimisation statements.

4.3 Abduction using ASP

We can represent any abductive task as an ASP.

The choice rule generates Answer Sets in which each of the **abducibles** is true. We then test each Answer Set using the **integrity constraints** and the constraint which says that the **observation** should be true.

The idea is that you generate lots of Answer Sets and then use constraints to test that they really are solutions to the problem you are solving

5 ASPAL

For a definite program P, as there is always a unique least Herbrand model $M(P)$. The task of ILP is to find a hypothesis H such that $M(B \cup H)$ contains all of a set of positive examples and none of a set of negative examples.

5.1 Cautious Induction

Cautious induction is based on the cautious semantics of ASP.

Given background knowledge B, positive examples E^+ and negative examples E^-. Search for hypothesis H such that $B \cup H$ is satisfiable and **every** answer set of $B \cup H$ contains all positive examples but none of the negative examples.

We use $ILP_C < B,\ E^+,\ E^- >$ to denote the set of hypotheses which are cautious inductive solutions of the task $< B,\ E^+,\ E^- >$

5.1.1 Limitations

Consider the example

$$1\{value(C, heads),\ value(C, tails)\}1 :- coin(C)$$
$$coin(c_1)$$

The only atom that is true is **every** answer set is $coin(c_1)$, and so the hypothesis will only be $coin(c_1)$.

This is not what we are aiming for at all; we want to learn a program with two distinct Answer Sets corresponding to the coin being heads or tails. Cautious entailment of all examples in this case is too strong a requirement.

5.2 Brave Induction

Brave induction is based on the brave semantics of ASP.

Given background knowledge B, positive examples E^+ and negative examples E^-. Search for hypothesis H such that **at least one** answer set of $B \cup H$ contains all positive examples but none of the negative examples.

We use $ILP_B < B,\ E^+,\ E^- >$ to denote the set of hypotheses which are cautious inductive solutions of the task $< B,\ E^+,\ E^- >$

Two of the main non-monotonic ILP systems for brave induction tasks are: **XHAIL** and **ASPAL**

5.3 ASPAL

5.3.1 ASPAL Skeleton Rules

A skeleton rule for mode declaration $< M_h, M_b >$ is a compatible rule where all constants place markers are replaced with a different variable instead of a constant.

For every variable, ASPAL will add the type of the variable to the body of the rule. These additional atoms do not count towards the length of the rules.

Each skeleton rule represents a set of rules where the variables representing constants have been replaced with constants of the correct types.

Positive integers L_{max} and V_{max} specifies the number of literals allowed to appear in the body of the rule, and the number of unique variables allowed in the rule.

For example given the following mode declaration

$$modeh(1, penguin(+bird)) \qquad modeb(2, not\ can(+bird, \#ability))$$

And the back ground knowledge

$$B = \begin{cases} bird(a) \\ bird(b) \\ ability(fly) \\ ability(swim) \\ can(a, fly) \\ can(b, swim) \end{cases}$$

$L_{max} = 2$ and $V_{max} = 1$. The set of skeleton rules are

$$S_M = \begin{cases} penguin(V1) :- bird(V1) \\ penguin(V1) :- bird(V1),\ not\ can(V1, C1) \\ penguin(V1) :- bird(V1),\ not\ can(V1, C1),\ not\ can(V1, C2) \end{cases}$$

Which represents the set of rules

$$\begin{aligned} &penguin(V1) :- bird(V1) \\ &penguin(V1) :- bird(V1),\ not\ can(V1, fly) \\ &penguin(V1) :- bird(V1),\ not\ can(V1, swim) \\ &penguin(V1) :- bird(V1),\ not\ can(V1, fly),\ not\ can(V1, swim) \end{aligned}$$

5.3.2 ASPAL Encoding

Given S_M, B, E^+ and E^-, we can encode the search for inductive solution as ASP program.

For each skeleton rule R in the hypothesis space S_M, we add to literal $rule(R_{id}, C1, C2, ..., Cn)$ to the body.

$$S_M = \begin{cases} penguin(V1) :- bird(V1),\ rule(1) \\ penguin(V1) :- bird(V1),\ not\ can(V1, C1),\ rule(2, C1) \\ penguin(V1) :- bird(V1),\ not\ can(V1, C1),\ not\ can(V1, C2),\ rule(3, C1, C2) \end{cases}$$

The aggregate $\{rule(1),\ rule(2, fly),\ rule(2, swim),\ rule(3, fly, swim)\}$ causes one Answer Set to be generated for each hypothesis which could be constructed from the

skeleton rules.

We have to add the goal of the program as well as a constraint

$$goal : -e_1^+,\ e_2^+,...,\ e_n^+,\ not\ e_1^-,\ not\ e_2^-,...,\ not\ e_m^-$$
$$: -not\ goal$$

ASPAL uses an optimisation statement such that the optimal Answer Sets of the meta level program correspond exactly to the optimal inductive solutions of the task.

$$\#minimise[rule(R_1, c1, c2) = R_{length}, ...]$$

The length of the rule does not count any type predicate and the rule predicate, for example

$$\#minimise[rule(1) = 1,\ rule(2, fly) = 2,\ rule(2, swim) = 2,\ rule(3, fly, swim) = 3]$$

5.4 Limitations

Brave induction cannot be used to learn any hypothesis which only has constraints and does not generate anything new.

If a hypothesis is a solution from brave induction, there must be one answer set which contains all of the positive and none of the negative examples. Constraints will only rule out answer sets meaning this hypothesis is still a solution with or without the constraint.

6 Learning From Answer Sets

We have considered two different kinds of induction in ASP: brave and cautious.

We have seen the limitations of ASPAL is that it would not be able to learn the constraint which rule out Answer Sets.

We will look into another paradigm, **learning from answer sets**

6.1 Partial Interpretations

The examples in *learning from answer set* are partial interpretations.

Partial interpretation e is a pair of sets of atoms: inclusions e^{inc} and exculsions e^{exc}.

A Herbrand interpretation I extends a partial interpretation $e =< e^{inc},\ e^{exc} >$ if:

1. It includes all of the inclusions, $e^{inc} \subseteq I$
2. It includes none of the exclusions, $e^{exc} \cap I = \emptyset$

6.2 Learning From Answer Sets (LAS)

A Learning from Answer Set task is a tuple $< B,\ S_M,\ E^+,\ E^- >$.

B is an Answer Set program called the background knowledge

S_M is the set of rules with which we are allowed to construct our hypotheses. S_M is usually constructed from a set of mode declarations. LAS also allows hypothesis to be choice rules and constraints

E^+ and E^- are partial interpretations.

Hypothesis H is an inductive solution if

1. it is constructed from the rules in S_M, $H \subseteq S_M$
2. Every positive example in E^+ is extended by **at least one** Answer Set of $B \cup H$
3. No negative example in E^- is extended by any Answer Set of $B \cup H$

6.3 LAS and Brave Induction

A brave induction task is satisfied by a hypothesis H if and only if there is **at least one** Answer Set of $B \cup H$ which

1. Includes all of the positive examples E^+
2. None of the negative examples E^-

This is equivalent to have **at least one** Answer set of $B \cup H$ that extends the partial interpretation $< E^+,\ E^- >$.

This is equivalent to finding a hypothesis H such that there is **at least one** Answer Set of $B \cup H$ which covers one positive example $< E^+,\ E^- >$

$$ILP_b < B,\ E^+,\ E^- > \equiv ILP_{LAS} < B,\ S_M,\ \{< E^+,\ E^- >\},\ \emptyset >$$

6.4 LAS and Cautious Induction

A cautious induction task is satisfied by a hypothesis H if and only if **every** Answer Set of $B \cup H$ which

1. Includes all of the positive examples E^+
2. None of the negative examples E^-

This is equivalent to no Answer Set of $B \cup H$ that will contain any negative examples. And no Answer Set of $B \cup H$ that does not contain any positive examples.

That means there will be no Answer Set of $B \cup H$ that extends the partial interpretations $< \emptyset,\ e_1^+ >,\ldots,< \emptyset,\ e_m^+ >,\ < e_1^-,\ \emptyset >,\ldots,< e_n^-,\ \emptyset >$

$$ILP_b < B,\ \{e_1^+,\ldots,e_m^+\},\ \{e_1^-,\ldots,e_n^-\} > \equiv ILP_{LAS} < B,\ S_M,\ \emptyset,\ \{< \emptyset,\ e_1^+ >,\ldots,< \emptyset,\ e_m^+ >,< e_1^-,\ \emptyset >,\ldots,< e_n^-,\ \emptyset >\} >$$

7 Relevant Example, Context and Preference Learning

Learning from answer sets is capable of expressing both brave and cautious induction, and can learn programs with normal rules, choice rules and constraints.

ILASP1 and ILASP2 are slow. There are two ways in which we can make ILASP more efficient:

1. Relevant examples
2. Context-dependent examples

7.1 Relevant Examples

ILASP1 and 2 transform the entire learning task into a single ASP program, whose grounding is proportional in size to the number of examples. This makes it inefficient when there are many examples.

ILASP2i takes advantage of the fact that in real settings there is likely to be an overlap between different examples. If a set of examples is covered by exactly the same class of hypotheses, we need only consider a single example from this set.

In many tasks with a large number of examples, only a small subset of relevant examples are actually required. ILASP2i iteratively constructs a small subset of the examples which is representative of the full set of examples, at each stage using ILASP2 to find a hypothesis that covers the relevant examples.

In the worst case, the relevant example set will end up being equal to the full set of examples. In this case, ILASP2i will be slower than ILASP2. In most cases the final relevant subset is significantly smaller than the full set of examples.

7.2 Context Dependent Examples

The learning frameworks that we have seen so far cannot directly represent context-specific knowledge, meaning the background knowledge in each framework is fixed, and applies to every example.

Context dependent example allow us to express that the behaviour of the learned program should vary depending on the context.

A **context-dependent partial interpretation (CDPI)** is a pair $e =< e_{pi}, e_{ctx} >$:

1. e_{pi} is a partial interpretation
2. e_{ctx} is an ASP program called a context.

The **context** can be thought of as an extra bit of background knowledge that applies only to the current example.

Given a program P and interpretation I. I is an **accepting answer set** of e wrt P iff

1. I is an answer set of $P \cup e_{ctx}$ and

2. I extends e_{pi}

For example given $P = \{p :- not\ q\}$.

The interpretation $\{q\}$ is an accepting answer set of $<<\emptyset, \{p\}>, \{q\}>$ wrt P because $\{q\}$ is an answer set of $P \cup \{q\}$ and extends $<\emptyset, \{p\}>$.

The interpretation $\{p\}$ is **not** an accepting answer set of $<<\{p\}, \emptyset, >, \{q\}>$ wrt P because $\{p\}$ is not an answer set of $P \cup \{q\}$.

7.2.1 Context Depended Learning From Answer Set

Given a task T, we write $ILP^{context}_{LAS}(T)$ to denote the set of all inductive solutions of T.

The task is to find a hypothesis H such that:

1. For each positive example $<e_{pi}, e_{ctx}>$, there must be at least one answer set of $B \cup H \cup e_{ctx}$ that extends e_{pi}.

2. For each negative example $<e_{pi}, e_{ctx}>$, no answer set of $B \cup H \cup e_{ctx}$ should extend e_{pi}.

7.3 Preference Learning

This section explores the idea of representing and learning preferences under the answer set semantics. There are many different approaches to preference learning, two of the most common ones are **Collaborative filtering** and **Object ranking**

Collaborative filtering identify similar users, and one user is recommended an item based on the action of other users. Collaborative filtering is commonly used by recommender systems.

Object ranking aims to learn the ordering over a set of objects, based on examples of which objects are preferred to others.

In ASP, objects can be represented as answer sets. If one answer set A is more optimal than another B, then this indicates that A is preferred to B.

7.3.1 Weak Constraints

Optimisation statements create a preference ordering over the answer sets of any ASP program. A similar construct in ASP called a weak constraint serves the same purpose.

A weak constraint has the form

$$:\sim b_1,\ b_2,...,\ b_m.[wt@lev,\ t_1,\ t_2,...,\ t_n]$$

Where b are body literals. wt is the weight. lev is the priority level. Each t is a term.

For any ground program P and answer set A. $weak(P, A)$ is a set of **unique** tails whose body is satisfy by A. Because each tail is unique, the terms t is important.

Given a program P and answer set A and priority level lev.

$$score(P, lev, A) = \sum_{[wt@lev, t_1, t_2, ..., t_n] \in weak(P, A)} wt$$

Given two answer sets A and B, A is preferred to B iff for the **highest priority level** lev

$$score(P, lev, A) < score(P, lev, B)$$

7.3.2 Ordering Examples

An ordering example is a pair of CDPI $< e_1, e_2 >$. From the example we want to learn a weak constraint such that e_1 is preferred to e_2.

For an ordering to be **bravely respected**, there must be **at least one** pair of accepting answer sets A and B of e_1 and e_2 wrt $B \cup H$ such that A is preferred to B.

For an ordering to be **cautiously respected**, **every** pair of accepting answer sets A and B of e_1 and e_2 wrt $B \cup H$, A must be preferred to B.

8 Noisy Examples

In reality we cannot assume that all of the examples we are given will be perfectly labelled due to noise.

In real world datasets, where some examples are incorrectly labelled, there is often no single perfect hypothesis in the hypothesis space which covers all of the examples. In these cases, it would be useful to find a hypothesis that covers the majority of the examples.

8.0.1 Penalised Examples

Given an ILP framework ILP_B, an $n(ILP_B)$ task is an ILP_B such that every example is annotated with a weight, either an integer of infinity.

$$E^+ = \{q(1)@1, q(2)@1, ..., q(9)@1\}$$
$$E^- = \{q(10)@1\}$$

The idea is that a hypothesis doesn't necessarily have to cover all of the examples in order to be an inductive solution.

Given a task T and hypothesis H.

$$Score(H,\ T) = |H| + \sum_{e@w \in U} w$$

Where U is the set of examples in T that are **not covered** by H.

An inductive solution must have a finite score, meaning it must cover all examples with an infinite penalty. The optimal solution is a solution with **minimum score**.

The scoring function balances the sum of the weights of uncovered examples and the length of the hypothesis. This will prevent it from learning over complicated hypothesis.

The weights for each example can be seen as reliability. The lower the weight, the easier it is for that example to be ignored.

8.1 Penalised Brave Induction with ASP

We can make very minor changes to the ASPAL encoding to support penalised brave induction.

For an ILP_B task $< B,\ M,\ \{e_1^+,\ ...,\ e_m^+\},\ \{e_1^-,\ ...,\ e_n^-\} >$ contains the rule

$$\begin{aligned} goal &:- e_1^+,\ ...,\ e_m^+,\ not\ e_1^-,\ ...,\ not\ e_n^- \\ &:- not\ goal \end{aligned}$$

And also minimising statements such as

$$\#minimise\ [rule(1) = 1,\ rule(2, fly) = 2,\ rule(2, swim) = 2,\ rule(3, fly, swim)]$$

To $n(ILP_B)$ task $< B,\ M,\ \{e_1^+@w_1^+,\ ...,\ e_m^+@w_m^+\},\ \{e_1^-@w_1^-,\ ...,\ e_n^-@w_n^-\} >$. The goal rule is replaced with a set of weak constraints

$$\begin{aligned} &:\sim not\ e_1^+.[w_1^+@1,\ e_1^+] \\ &... \\ &:\sim not\ e_m^+.[w_m^+@1,\ e_m^+] \\ &:\sim e_1^-.[w_1^-@1,\ e_1^-] \\ &... \\ &:\sim e_n^-.[w_n^-@1,\ e_n^-] \end{aligned}$$

The optimisation statement is also replaced with a set of weak constraints

$$\begin{aligned} &:\sim rule(1).[1@1,\ rule(1)] \\ &:\sim rule(2, A).[2@1,\ rule(2, A)] \\ &:\sim rule(3, A, B).[3@1,\ rule(3, A, B)] \end{aligned}$$

8.1.1 Penalised Brave Induction to Brave Induction

A penalised brave induction task can be translated into a non-noisy brave task using the following steps:

1. Assign an identifier e_{id} to each example e.
2. Add a fact $noise(e_{id})$ to the hypothesis space $S_M$for each e.
3. Add the rules $c(e_{id}) \leftarrow noise(e_{id})$ and $c(e_{id}) \leftarrow e$ for each positive example to B
4. Add the rule $c(e_{id}) \leftarrow not\ noise(e_{id}),\ e$ for each negative example to B
5. Replace E^+ with $\{c(e_{id}) | e \in E^+\}$
6. Replace E^- with $\{c(e_{id}) | e \in E^-\}$

A hypothesis that doesn't cover a particular example e must include a fact $noise(e_{id})$, which has the effect of raising the length of the hypothesis by 1.

This transformation assumes that the weights of each examples is 1.

Penalised Brave Induction Task

$$
\begin{aligned}
B &= \{t(1...10)\} \\
S_M &= \{q(X) \leftarrow t(X)\} \\
E^+ &= \{q(1)@1,\ ...,\ q(9)@1\} \\
E^- &= \{q(10)@1\}
\end{aligned}
$$

Brave Induction Task

$$
B = \begin{cases}
t(1...10) & \\
c(1) \leftarrow q(1) & c(1) \leftarrow noise(1) \\
c(2) \leftarrow q(2) & c(2) \leftarrow noise(2) \\
... & \\
c(9) \leftarrow q(9) & c(9) \leftarrow noise(9) \\
c(10) \leftarrow q(10),\ not\ noise(10) &
\end{cases}
$$

$$
S_M = \begin{cases}
q(X) \leftarrow t(X) \\
noise(1) \\
noise(2) \\
... \\
noise(10)
\end{cases}
$$

$$
\begin{aligned}
E^+ &= \{c(1),\ c(2),...,\ c(9)\} \\
E^- &= \{c(10)\}
\end{aligned}
$$

8.1.2 Penalised Brave Induction to Penalised LAS

1. Translate the penalised brave task into a non-noisy brave task

2. Translate non-noisy brave task to non-noisy LAS with one positive example

3. Translate non-noisy LAS to penalised LAS by setting the penalty of the only positive example to be infinity

8.2 Penalised Cautious Induction to Penalised LAS

There is a more natural translation from penalised cautious induction to penalised LAS. This is because the standard translation from cautious induction to LAS yields a task with only a single inclusion or exclusion in each partial interpretation.

Penalised Cautious Induction Task

$$\begin{aligned} B &= \{t(1...10)\} \\ S_M &= \{q(X) \leftarrow t(X)\} \\ E^+ &= \{q(1)@1,\ ...,\ q(9)@1\} \\ E^- &= \{q(10)@1\} \end{aligned}$$

Penalised LAS Inductive Task

$$\begin{aligned} B &= \{t(1...10)\} \\ S_M &= \{q(X) \leftarrow t(X)\} \\ E^+ &= \{<\emptyset,\ \emptyset>\} \\ E^- &= \{<\emptyset,\ \{q(1)@1\}>, ..., <\emptyset,\ \{q(9)@1\}>, <\{q(10)@1\},\ \emptyset>\} \end{aligned}$$

Note that the empty positive example has an infinite penalty

9 Probabilistic Logic Programming

So far, we have been concentrating on notions of inference and structural learning that are underpinned by the principle of exact truth, where knowledge is assumed to be either true or false. But real-life situations are often uncertain

Probabilistic Logic Programming is part of a bigger area of work, called Probabilistic Programming, which focuses on the development of systems that can help make decisions in the face of uncertainty. A key representative of these languages is ProbLog.

Intuitively, probabilistic logic programming assumes probability on facts. In pure Datalog a set of facts F defines a precise scenario or single world. By assigning probability to the facts we are essentially indicating the existence of several possible worlds.

9.1 Possible World

Given the follow probabilistic facts:

$$0.8 :: stress(ann).$$

$$0.6 :: influences(ann, bob)$$
$$0.2 :: influences(bob, carl)$$

Different choices of Boolean assignment to the probabilistic random variable give different possible worlds. If we have n probabilistic random variable, then there are 2^n possible worlds.

We can make an atomic random choice on each fact in F. This will construct a vector ω known as a **possible world**.

In the example above, one possible world can be $\omega = \langle true, false, true \rangle$

9.1.1 Probability of Possible World

Given a set F' of n probabilistic atoms, we can then define the notion of composite choice K' over the set F' as a set of atomic choices for the atoms in F':

$$K' = (F_1, x_1), (F_2, x_2), ..., (F_n, x_n)$$

where x_i is either true or false. The probability of a world is calculated by

$$P(K') = \prod_{(F_i, true) \in K'} P_i \times \prod_{(F_i, false) \in K'} (1 - P_i)$$

9.1.2 Probability Measure

The **Sample Space** is defined as all the possible world constructed from F

$$W_f = \{w_1, w_2, ..., w_n\}$$

The **measurable space** is defined as (W_F, Ω_F). Where Ω_F is the power set of W_F

Given a measurable space (W_F, Ω_F), we can define a probability measure μ of any set σ possible worlds as the sum of the probability of the possible worlds that belong to σ:

$$\mu(\sigma_i) = \sum_{w_j \in \sigma_i} P(w_j)$$

For example $\mu(\{\omega_1, \omega_2, ..., \omega_4\}) = P(\omega_1) + P(\omega_2) + ... + P(\omega_4)$

9.1.3 LHM for Each world

We can construct a least herbrand model for each world, denote as $M_P(\omega_i)$

The probability distribution over possible worlds can be extended to a probability distribution over all the atoms that are in the signature of a given probabilistic logic program. Given any ground atom q

$$P(q) = \sum_{\omega_i \models q} P(\omega_i)$$

Similarly, given the conjuncture of ground atoms q_1, q_2, ..., q_n

$$P(q_1, q_2, ..., q_n) = \sum_{\omega_i \models q_1 \wedge q_2 \wedge ... \wedge q_n} P(\omega_i)$$

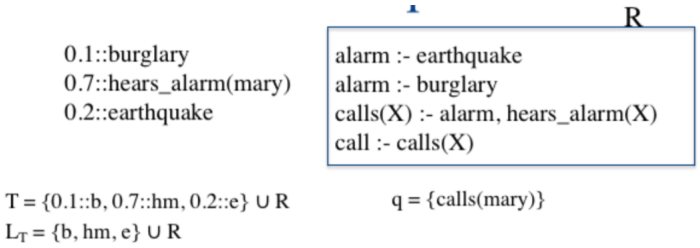
9.2 ProbLog

ProbLog program T is a set of definite clauses (possibly) labelled with a probability.

L_T the set of all clauses in T without the probability label.

The probability of a given sampled program L is calculated as the product of the probability of the clauses that have been sampled in L, and 1 minus the probability of the clauses in T that have not been sampled in L.

Given a query q, it can be proved from many of the sampled programs $L \subseteq L_T$. For each sample program, there will be a relevant ground instantiation of L denoted as M.



Given the example above, the query q can be proved from few sampled programs

$$L_1 = \{b, hm\} \cup R$$
$$L_2 = \{e, hm\} \cup R$$
$$L_3 = \{b, e, hm\} \cup R$$

To calculate the probability of the query $calls(mary)$ given the program T.

$$\begin{aligned} P(calls(mary)|T) &= \sum_{L_i \subseteq L_T} P(calls(mary), L_i|T) \\ &= \sum_{L_i \subseteq L_T} P(calls(mary)|L_i) \times P(L_i|T) \end{aligned}$$

$P(calls(mary)|L_i) = 1$ for L_1, L_2 and L_3. And $P(calls(mary)|L_i) = 0$ otherwise. Therefore

$$\begin{aligned} P(calls(mary)|T) &= P(L_1|T) + P(L_2|T) + P(L_3|T) \\ &= 0.1 \times 0.7 \times 0.8 + 0.2 \times 0.7 \times 0.9 + 0.2 \times 0.7 \times 0.1 = 0.196 \end{aligned}$$

9.3 Computation of success query

The method consists of two steps:

1. A proof tree for query q is constructed using only L_T. This steps generates a DNF formula, where each disjunct corresponds to a successful branch of the proof tree. Each disjunct is a conjunction of ground instances of the probabilistic clauses (or facts) used in the branch derivation.

2. Expressed the DNF into Binary Decision Diagram (BDD) to compute probability of query

9.3.1 Binary Decision Diagram (BDD)

A binary decision tree takes in input a set of Boolean variables $x_1, ..., x_n$. Each node has two subtrees, one where $x_i = 0$ and one where $x_i = 1$. At the leaves we get either 0 or 1, which is the output of the function on the inputs that constitute the path from the root to the leaf.

If we test variables in a fixed order $x_1, ..., x_n$, then the binary decision tree is unique.

Computing Probability from BDD

Algorithm: Probability calculation for BDDs

PROBABILITY(input: BDD node n)

If *n* is the 1-terminal **then return 1**
If *n* is the 0-terminal **then return 0**
Let p_n be the probability of the clause represented by *n*'s random variable
Let *h* and *l* be the high and low children of *n*
prob(h):= PROBABILITY(*h*)
prob(l) := PROBABILITY(*l*)
return p_n x *prob(h)* + (1 - p_n) x *prob(l)*

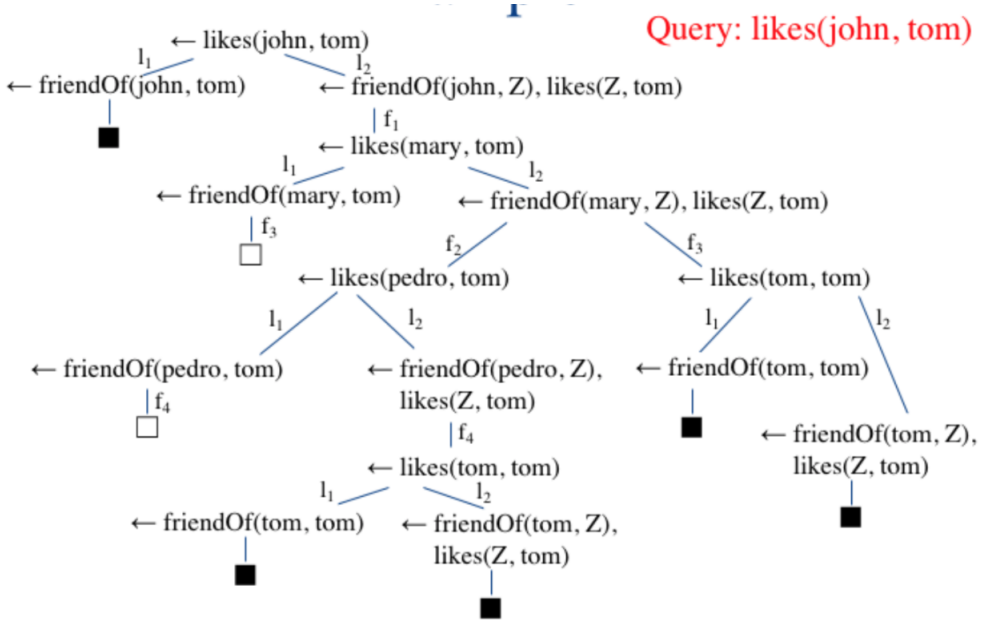
Child under 1 edge

Child under 0 edge

9.4 Example

T
1.0:: likes(X,Y) :- friendOf(X,Y).
0.8:: likes(X,Y) :- friendOf(X,Z), likes(Z,Y).
0.5:: friendOf(john, mary).
0.5:: friendOf(mary, pedro).
0.5:: friendOf(mary, tom).
0.5:: friendOf(pedro, tom).

Given the query $likes(john, tom)$. The first step is to construct a proof tree.



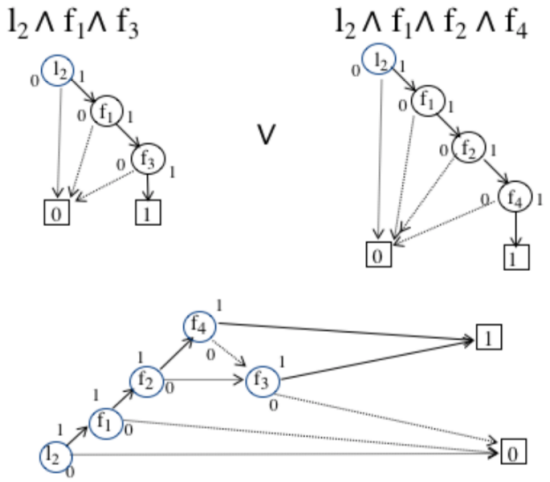
The DNF corresponding to the proof tree is

$$(l_1 \wedge l_2 \wedge f_1 \wedge f_3) \vee (l_2 \wedge f_1 \wedge f_2 \wedge l_1 \wedge f_4)$$

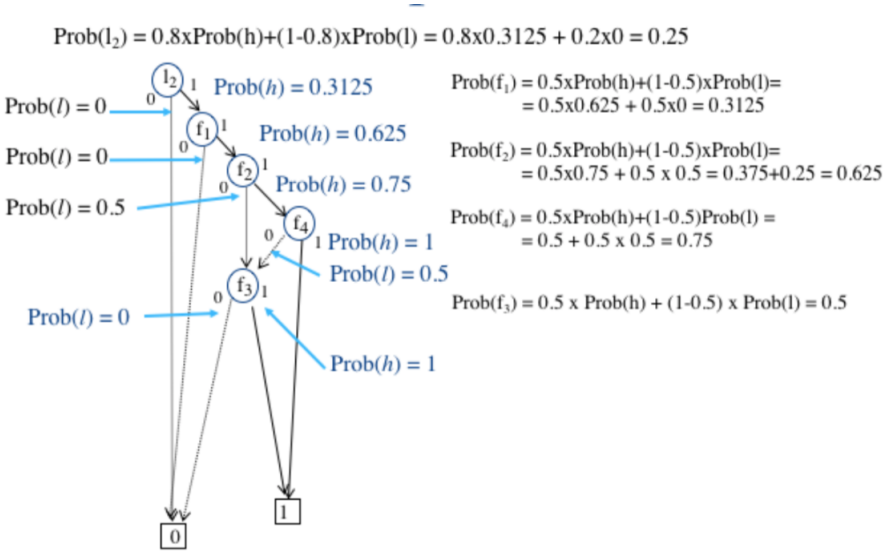
Since probability of l_1 is 1, we can simplify it to

$$(l_2 \wedge f_1 \wedge f_3) \vee (l_2 \wedge f_1 \wedge f_2 \wedge f_4)$$

The next step is to construct a BDD for the DNF



Which can be used to calculate the probability


Prob(l2) = 0.8xProb(h)+(1-0.8)xProb(l) = 0.8x0.3125 + 0.2x0 = 0.25
Prob(h) = 0.3125
Prob(l) = 0
Prob(h) = 0.625
Prob(l) = 0
Prob(h) = 0.75
Prob(l) = 0.5
Prob(h) = 1
Prob(l) = 0.5
Prob(l) = 0
Prob(h) = 1
Prob(f1) = 0.5xProb(h)+(1-0.5)xProb(l)=
= 0.5x0.625 + 0.5x0 = 0.3125
Prob(f2) = 0.5xProb(h)+(1-0.5)xProb(l)=
= 0.5x0.75 + 0.5 x 0.5 = 0.375+0.25 = 0.625
Prob(f4) = 0.5xProb(h)+(1-0.5)Prob(l) =
= 0.5 + 0.5 x 0.5 = 0.75
Prob(f3) = 0.5 x Prob(h) + (1-0.5) x Prob(l) = 0.5