Advances in Cognitive Systems X (20XX) 1-6 Submitted X/20XX; published X/20XX

Details of the Hypothesis Spaces used in "Inductive Learning of Answer Set Programs from Noisy Examples"

Mark Law MARK.LAW09@IMPERIAL.AC.UK
Alessandra Russo A.RUSSO@IMPERIAL.AC.UK
Krysia Broda K.BRODA@IMPERIAL.AC.UK

Abstract

In this document we give details of the hypothesis spaces omitted from the paper "Inductive Learning of Answer Set Programs from Noisy Examples". In ILASP tasks, hypothesis spaces can either be given in full, or defined by a mode bias (similarly to other ILP systems). In this document we formalise ILASP's particular form of mode bias and present hypothesis spaces used in the experiments in Section 4 of the paper.

1. The ILASP mode bias

Definition 1.1 formalises the notion of mode bias used by our ILASP systems.

Definition 1.1. An *ILASP mode bias* M is a set of meta-level atoms, which take the following forms:

- *Mode declarations*, each of which is a meta-level atom of the form $\texttt{class}(\texttt{r}, \texttt{a}, (\texttt{opt}_1, \ldots, \texttt{opt}_\texttt{n}))$, such that:
  - r is an integer called the *recall*. If r is omitted, this corresponds to an unrestricted (infinite) recall.
  - $\texttt{class} \in \{\#\texttt{modeh}, \#\texttt{modeha}, \#\texttt{modeb}, \#\texttt{modeo}\}$.
  - $\texttt{a}$ is a term (representing a reified atom) which may contain *placeholder* arguments of the form $\texttt{var}(\texttt{t})$ or $\texttt{const}(\texttt{t})$, where $\texttt{t}$ is a constant called the *type* of the argument.
  - $\texttt{opt}_1, \ldots, \texttt{opt}_\texttt{n}$ is a list of constants called *options*. If $n = 0$, the list of options is omitted.
- *Constant declarations*, each of which is a meta-level atom of the form $\#\texttt{constant}(\texttt{t}, \texttt{c})$ where $\texttt{t}$ and $\texttt{c}$ are both constants ($\texttt{t}$ is called a type).
- *Weight declarations*, each of which is a meta-level atom of the form $\#\texttt{weight}(\texttt{c})$ where $\texttt{c}$ is a constant.

- *Parameters* of the form $\#\texttt{param}(\texttt{i})$, where $\#\texttt{param} \in \{\#\texttt{minhl}, \#\texttt{maxhl}, \#\texttt{maxv}, \#\texttt{maxp}, \#\texttt{maxbl}, \#\texttt{maxrl}\}$[1] and i is an integer. There is at most one atom in M per $\#\texttt{param}$ predicate.
- The *flag* $\{\#\texttt{no\_constraints}\}$.

An ILASP mode bias can be used to specify a full hypothesis space. Definition 1.3 formalises the notion of the expansion of a set of mode declarations. We first formalise the notion of an atom being an *instance* of a mode declaration in Definition 1.2. Each variable in a rule is assigned a constant called its *type*. These types correspond to the types in the variable placeholders. No variable is allowed to take more than one type.

Definition 1.2. Given an ILASP mode bias M, let $\texttt{m}$ be a mode declaration $\texttt{c}(\texttt{r}, \texttt{a}, (\texttt{opt}_1, \ldots, \texttt{opt}_\texttt{n})) \in M$. An atom $\texttt{a}'$ is an *instance* of $\texttt{m}$ if each of the following conditions hold:

- $\texttt{a}'$ can be constructed from $\texttt{a}$ by replacing each placeholder $\texttt{var}(\texttt{t})$ is with a variable of type $\texttt{t}$ and each placeholder $\texttt{const}(\texttt{t})$ with a constant $\texttt{c}$ such that $\#\texttt{constant}(\texttt{t}, \texttt{c}) \in M$.
- If $\texttt{anti\_reflexive} \in \{\texttt{opt}_1, \ldots, \texttt{opt}_\texttt{n}\}$ and $\texttt{a}'$ has arity 2, then its two arguments are non-equal.
- If $\texttt{symmetric} \in \{\texttt{opt}_1, \ldots, \texttt{opt}_\texttt{n}\}$ and $\texttt{a}'$ has exactly two arguments $\texttt{arg}_1$ and $\texttt{arg}_2$, then $\texttt{arg}_1 \prec \texttt{arg}_2$ (where $\prec$ is a lexicographical ordering of all possible terms).

If $\texttt{a}'$ is an instance of $\texttt{m}$ and $\texttt{positive} \notin \{\texttt{opt}_1, \ldots, \texttt{opt}_\texttt{n}\}$, then $\texttt{not a}'$ is also an instance of $\texttt{m}$.

Note that unlike the mode declarations in other systems, there is no need for the types to correspond to predicates in the background knowledge.

Definition 1.3. Given a mode bias M, we define the rules that *conform to* M as follows:

- A conjunction of naf-literals C is said to be valid over a class $\texttt{c} \in \{\#\texttt{modeb}, \#\texttt{modeo}\}$ given M iff each of the following conditions hold:
  1. Each literal in C is an instance of at least one mode declaration in M with class $\texttt{c}$.
  2. If there is a parameter atom $\#\texttt{maxv}(\texttt{i}) \in M$, then the number of variables in C is at most $\texttt{i}$.
  3. If there is a parameter atom $\#\texttt{maxbl}(\texttt{i}) \in M$, then the number of literals in C is at most $\texttt{i}$.
  4. Every variable in C occurs in at least one positive literal in C (i.e. C is *safe*).
- A hard constraint R is said to conform to M iff $\#\texttt{no\_constraints} \notin M$ and $body(R)$ is a valid conjunction over $\#\texttt{modeb}$ given M.

1. In the ILASP implementation, $\#\texttt{maxbl}$ and $\#\texttt{maxrl}$ are passed to the system as the command line flags `-ml=[integer]` and `-max-rule-length=[integer]`, respectively.

- A normal rule R is said to conform to M iff each of the following conditions hold:
  1. $body(R)$ is a valid conjunction over $\#\texttt{modeb}$ given M,
  2. $head(R)$ is an instance of at least one mode declaration in M with class $\#\texttt{modeh}$.
  3. each variable in $head(R)$ occurs in $body(R)$.
  4. If there is a parameter atom $\#\texttt{maxrl(i)} \in M$, then the number of literals in R is at most $\texttt{i}$.
- A choice rule R is said to conform to M iff each of the following conditions hold:
  1. $body(R)$ is a valid conjunction over $\#\texttt{modeb}$ given M.
  2. each atom in $heads(R)$ is an instance of at least one mode declaration in M with class $\#\texttt{modeha}$.
  3. If there is a parameter atom $\#\texttt{minhl(i)} \in M$, then $|heads(R)| \geq \texttt{i}$.
  4. If there is a parameter atom $\#\texttt{maxhl(i)} \in M$, then $|heads(R)| \leq \texttt{i}$.
  5. If there is a parameter atom $\#\texttt{maxrl(i)} \in M$, then the number of literals in $R \leq \texttt{i}$.
  6. The lower bound $\texttt{lb}$ and upper bound $\texttt{ub}$ of $head(R)$ are such that $0 \leq \texttt{lb} \leq \texttt{ub} \leq |heads(R)|$.
  7. All variables in $head(R)$ occur in $body(R)$.
- A weak constraint W with the tail $[\texttt{wt@lev}, \texttt{t}_1, \ldots, \texttt{t}_\texttt{n}]$ is said to conform to M iff each of the following conditions hold:
  1. $body(W)$ is a valid conjunction over $\#\texttt{modeo}$ given M.
  2. $\texttt{wt}$ is either an integer such that $\#\texttt{weight(wt)} \in M$ or a variable $\texttt{V}$ that occurs in $body(W)$ (or the negation $-\texttt{V}$ of such a variable) such that $\texttt{weight(t)} \in M$, where $\texttt{t}$ is the type of $\texttt{V}$.
  3. $\texttt{lev}$ is a positive integer and if there is a parameter atom $\#\texttt{maxp(i)} \in M$, then $\texttt{lev} \leq \texttt{i}$.
  4. $\texttt{t}_1 = W_{id}$ (the unique identifier of W).
  5. $\texttt{t}_2, \ldots, \texttt{t}_\texttt{n}$ is the list of variables that occur in $body(W)$.

A hypothesis space S_M is called an *expansion* of S_M if each rule in S_M conforms to M, and every rule that conforms to M is strongly equivalent to at least one rule in S_M.

We use S_M to denote an arbitrary (finite) expansion of M[2].

2. Note that there is guaranteed to be at least one finite expansion of any finite mode bias M for which M contains values for each of the parameters (the ILASP implementation assigns default values if these are unspecified). This is because there must be a finite set of (non-equivalent) valid conjunctions of any length given M. This fixes the set of possible bodies, and as the variables in the rest of a rule must all occur in the body of the rule, this means there is a finite set of non-equivalent rules that conform to M.

In addition to specifying hypothesis spaces through mode biases, ILASP also allows for hypothesis spaces to be given in full, as a set of rule declarations of the form $L \sim R$, where L is a positive integer and R is an ASP rule. L is set as the length of R (denoted $|R|$).

The latest versions of the ILASP system also support *bias constraints*, which are used to cut rules out of the hypothesis space. Rather than fully specify the semantics of these bias constraints, we have commented on the effect of any bias constraints that we use in language biases throughout the rest of the section.

2. Language Biases Used in the Evaluation Sections

2.1 Hamilton Graphs

The Hamilton problem was first run before bias constraints were implemented in ILASP. We used a mode bias to generate a hypothesis space and then cut out rules which were equivalent to other rules in the hypothesis space (or at least were guaranteed to give exactly the same results), or which were guaranteed not to appear in an optimal inductive solution of the task. In modern versions of ILASP, we could achieve similar results using bias constraints. The original language bias was:

```
#modeha(1, in(var(node1),var(node1))).
#modeb(1, edge(var(node1), var(node1)), (positive)).

#modeb(2, in(var(node), var(node)), (positive, anti_reflexive)).
#modeb(1, var(node) != var(node), (positive, symmetric, anti_reflexive)).

#modeh(1, reach(var(node))).
#modeb(1, reach(var(node))).
#modeb(1, node(var(node)), (positive)).
#modeb(1, in(const(node),var(node)), (positive)).
#constant(node, 1).

#maxhl(1).
```

The final set of rules (where $n \sim R$ denotes that R is in the hypothesis space, and $|R| = n$) were:

```
1 ~  :- edge(V0, V0).
1 ~  :- edge(V0, V1).
1 ~  :- in(1,V0).
```

```
1 ~  :- in(V0,V1).
1 ~  :- reach(V0).
2 ~  :- edge(V0, V0), in(1,V1).
2 ~  :- edge(V0, V0), in(V1,V2).
2 ~  :- edge(V0, V0), reach(V1).
2 ~  :- edge(V0, V1), in(1,V2).
2 ~  :- edge(V0, V1), reach(V2).
2 ~  :- in(1,V0), reach(V0).
2 ~  :- in(1,V0), reach(V1).
2 ~  :- in(V0,V1), V0 != V1.
2 ~  :- in(V0,V1), in(1,V0).
2 ~  :- in(V0,V1), in(1,V1).
2 ~  :- in(V0,V1), in(1,V2).
2 ~  :- in(V0,V1), in(V0,V2).
2 ~  :- in(V0,V1), in(V1,V2).
2 ~  :- in(V0,V1), reach(V0).
2 ~  :- in(V0,V1), reach(V1).
2 ~  :- in(V0,V1), reach(V2).
2 ~  :- node(V0), not reach(V0).
2 ~ reach(V0) :- in(1,V0).
2 ~ reach(V0) :- in(V0,V1).
2 ~ reach(V1) :- in(V0,V1).
3 ~  :- edge(V0, V0), in(1,V1), reach(V1).
3 ~  :- edge(V0, V0), in(1,V1), reach(V2).
3 ~  :- edge(V0, V0), in(V1,V2), V1 != V2.
3 ~  :- edge(V0, V0), in(V1,V2), in(1,V1).
3 ~  :- edge(V0, V0), in(V1,V2), in(1,V2).
3 ~  :- edge(V0, V0), in(V1,V2), reach(V1).
3 ~  :- edge(V0, V0), in(V1,V2), reach(V2).
3 ~  :- edge(V0, V0), node(V1), not reach(V1).
3 ~  :- edge(V0, V1), in(1,V2), reach(V2).
3 ~  :- edge(V0, V1), node(V2), not reach(V2).
3 ~  :- in(1,V0), node(V0), not reach(V0).
3 ~  :- in(1,V0), node(V1), not reach(V1).
3 ~  :- in(V0,V1), V0 != V1, in(1,V0).
3 ~  :- in(V0,V1), V0 != V1, in(1,V1).
3 ~  :- in(V0,V1), V0 != V1, in(1,V2).
3 ~  :- in(V0,V1), V0 != V1, reach(V0).
3 ~  :- in(V0,V1), V0 != V1, reach(V1).
3 ~  :- in(V0,V1), V0 != V1, reach(V2).
3 ~  :- in(V0,V1), in(1,V0), reach(V0).
3 ~  :- in(V0,V1), in(1,V0), reach(V1).
3 ~  :- in(V0,V1), in(1,V0), reach(V2).
3 ~  :- in(V0,V1), in(1,V1), reach(V0).
```

```
3 ~  :- in(V0,V1), in(1,V1), reach(V1).
3 ~  :- in(V0,V1), in(1,V1), reach(V2).
3 ~  :- in(V0,V1), in(1,V2), reach(V0).
3 ~  :- in(V0,V1), in(1,V2), reach(V1).
3 ~  :- in(V0,V1), in(1,V2), reach(V2).
3 ~  :- in(V0,V1), in(V0,V2), V0 != V1.
3 ~  :- in(V0,V1), in(V0,V2), V0 != V2.
3 ~  :- in(V0,V1), in(V0,V2), V1 != V2.
3 ~  :- in(V0,V1), in(V0,V2), in(1,V0).
3 ~  :- in(V0,V1), in(V0,V2), in(1,V1).
3 ~  :- in(V0,V1), in(V0,V2), in(1,V2).
3 ~  :- in(V0,V1), in(V0,V2), reach(V0).
3 ~  :- in(V0,V1), in(V0,V2), reach(V1).
3 ~  :- in(V0,V1), in(V0,V2), reach(V2).
3 ~  :- in(V0,V1), in(V1,V2), V0 != V1.
3 ~  :- in(V0,V1), in(V1,V2), V0 != V2.
3 ~  :- in(V0,V1), in(V1,V2), V1 != V2.
3 ~  :- in(V0,V1), in(V1,V2), in(1,V0).
3 ~  :- in(V0,V1), in(V1,V2), in(1,V1).
3 ~  :- in(V0,V1), in(V1,V2), in(1,V2).
3 ~  :- in(V0,V1), in(V1,V2), reach(V0).
3 ~  :- in(V0,V1), in(V1,V2), reach(V1).
3 ~  :- in(V0,V1), in(V1,V2), reach(V2).
3 ~  :- in(V0,V1), node(V0), not reach(V0).
3 ~  :- in(V0,V1), node(V1), not reach(V1).
3 ~  :- in(V0,V1), node(V2), not reach(V2).
3 ~  :- reach(V0), node(V0), not reach(V0).
3 ~  :- reach(V0), node(V1), not reach(V1).
3 ~ 0 {in(V0,V0) } 1 :- edge(V0, V0).
3 ~ 0 {in(V0,V0) } 1 :- edge(V0, V1).
3 ~ 0 {in(V0,V1) } 1 :- edge(V0, V1).
3 ~ 0 {in(V1,V0) } 1 :- edge(V0, V1).
3 ~ 0 {in(V1,V1) } 1 :- edge(V0, V1).
3 ~ reach(V0) :- in(1,V0), reach(V1).
3 ~ reach(V0) :- in(V0,V1), V0 != V1.
3 ~ reach(V0) :- in(V0,V1), in(1,V0).
3 ~ reach(V0) :- in(V0,V1), in(1,V1).
3 ~ reach(V0) :- in(V0,V1), in(1,V2).
3 ~ reach(V0) :- in(V0,V1), in(V0,V2).
3 ~ reach(V0) :- in(V0,V1), in(V1,V2).
3 ~ reach(V0) :- in(V0,V1), reach(V1).
3 ~ reach(V0) :- in(V0,V1), reach(V2).
3 ~ reach(V1) :- edge(V0, V0), in(1,V1).
3 ~ reach(V1) :- edge(V0, V0), in(V1,V2).
```

```
3 ~ reach(V1) :- in(V0,V1), V0 != V1.
3 ~ reach(V1) :- in(V0,V1), in(1,V0).
3 ~ reach(V1) :- in(V0,V1), in(1,V1).
3 ~ reach(V1) :- in(V0,V1), in(1,V2).
3 ~ reach(V1) :- in(V0,V1), in(V0,V2).
3 ~ reach(V1) :- in(V0,V1), in(V1,V2).
3 ~ reach(V1) :- in(V0,V1), reach(V0).
3 ~ reach(V1) :- in(V0,V1), reach(V2).
3 ~ reach(V2) :- edge(V0, V0), in(V1,V2).
3 ~ reach(V2) :- edge(V0, V1), in(1,V2).
3 ~ reach(V2) :- in(V0,V1), in(1,V2).
3 ~ reach(V2) :- in(V0,V1), in(V0,V2).
3 ~ reach(V2) :- in(V0,V1), in(V1,V2).
```

2.2 Journey Preferences

```
#modeo(1,leg_mode(var(leg), const(mode_of_transport)), (positive)).
#modeo(1,leg_distance(var(leg), var(distance)), (positive)).
#modeo(1,leg_crime_rating(var(leg), var(crime_rate)), (positive)).
#modeo(1, var(crime_rate) > const(crime_rate)).
#weight(distance).
#weight(1).
#constant(mode_of_transport, walk).
#constant(mode_of_transport, bus).
#constant(mode_of_transport, bicycle).
#constant(mode_of_transport, car).
#constant(crime_rate, 1).
#constant(crime_rate, 2).
#constant(crime_rate, 3).
#constant(crime_rate, 4).
#maxp(3).
#maxv(2).
```

2.3 Predecessor

```
#modeh(predecessor(var(any),var(any))).
#modeb(succ(var(any),var(any))).
#modeb(number(var(any)), (positive)).
#maxv(2).
#maxbl(3).
```

2.4 Less Than

```
#modeh(less_than(var(any), var(any)), (positive)).
#modeb(1, succ(var(any), var(any)), (positive)).
#modeb(1, less_than(var(any), var(any)), (positive)).
#maxbl(2).
```

2.5 Member

```
#modeh(member(var(number), var(list))).
#modeh(p(var(number), var(list))).
#modeb(1, p(var(number), var(list))).
#modeb(1, value(var(list), var(number)), (positive)).
#modeb(1, cons(var(list), var(list)), (positive)).
#modeb(list(var(list)), (positive)).
#modeb(number(var(number)), (positive)).
#maxbl(5).

% enforces that the list/number nodes are used as typing atoms (i.e. they occur
% if and only if a variable of that type occurs).
#bias(":- body(p(V1, V2)), not body(number(V1)).").
#bias(":- body(p(V1, V2)), not body(list(V2)).").
#bias(":- body(value(V1, V2)), not body(list(V1)).").
#bias(":- body(value(V1, V2)), not body(number(V2)).").
#bias(":- body(cons(V1, V2)), not body(list(V1)).").
#bias(":- body(cons(V1, V2)), not body(list(V2)).").
#bias(":- body(number(V1)), not body(p(V1, _)), not body(naf(p(V1, _))),
          not body(naf(value(_, V1))), not body(value(_, V1)).").
#bias(":- body(list(V1)), not body(p(_, V1)), not body(naf(p(_, V1))),
          not body(naf(value(V1, _))), not body(value(V1, _)),
          not body(cons(V1, _)), not body(cons(_, V1)).").
#bias(":- head(p(_, _)), body(naf(p(_, _))).").
```

2.6 Connected

```
#modeh(connected(var(any), var(any)), (positive)).
#modeb(1, edge(var(any), var(any)), (positive)).
#modeb(1, connected(var(any), var(any)), (positive)).
#maxbl(2).
```

2.7 Undirected Edge

```
#modeh(undirected_edge(var(node), var(node))).
#modeb(edge(var(node), var(node))).
#modeb(node(var(node)), (positive)).
#maxbl(5).

% enforces that the node atoms are used as typing atoms (i.e. they occur if and
% only if a variable of that type occurs).
#bias(":- body(edge(V1, V2)), not body(node(V1)).").
#bias(":- body(edge(V1, V2)), not body(node(V2)).").
#bias(":- body(naf(edge(V1, V2))), not body(node(V1)).").
#bias(":- body(naf(edge(V1, V2))), not body(node(V2)).").
#bias(":- body(node(V)), not body(edge(V, _)), not body(edge(_, V)),
          not body(naf(edge(V, _))), not body(naf(edge(_, V))).").
```

2.8 Sentence Chunking

```
#constant(postype, c). % for each part of speach tag that occurs in the task.

#modeh(split(var(token))).
#modeb(1, pos(const(postype),var(token)), (positive)).
#modeb(1, prevpos(const(postype),var(token)), (positive)).
#modeb(1, test_split(var(token)), (positive)).
#maxv(1).

% This constraint means that test_split(V0) must occur in the body of every
% rule. As each context contains test_split(X) for each X such that split(X)
% occurs in the inclusions or exclusions of the corresponding partial
% interpretation, this just means that the only ground instances of rules in
% the hypothesis space that are considered are those that could affect whether
% or not the example is covered.
#bias(":- not body(test_split(_))."). 
```

2.9 Cars

```
#weight(cap).
#weight(1).
#weight(-1).
#modeo(1,body(const(bool)), (positive)).
#modeo(1,transmission(const(bool)), (positive)).
#modeo(1,fuel(const(bool)), (positive)).
#modeo(1,engine_cap(var(cap))).
#constant(bool, 1).
#constant(bool, 2).
#maxp(5).
```

2.10 SUSHI

```
#modeo(1, value(const(val),var(val))).
#modeo(1, minor_group(const(mg)), (positive)).
#modeo(1, seafood, (positive)).
#modeo(1, maki, (positive)).
#maxp(5).
#weight(val).
#weight(1).
#weight(-1).
#constant(val, oil).
#constant(val, price).
#constant(val, freq).
#constant(val, freq2).

% The constants for minor groups 2, 4 and 10 were omitted as none of the
% sushi's in this part of the dataset have any of these minor groups.
#constant(mg, 1).
```

```
#constant(mg, 3).
#constant(mg, 5).
#constant(mg, 6).
#constant(mg, 7).
#constant(mg, 8).
#constant(mg, 9).
#constant(mg, 11).
```