Mach Learn (2018) 107:1141–1169
https://doi.org/10.1007/s10994-018-5708-2

Best-effort inductive logic programming via fine-grained cost-based hypothesis generation

The Inspire system at the inductive logic programming competition

Peter Schüller[1,2] · **Mishal Benz**[3,4]

Received: 31 March 2017 / Accepted: 5 April 2018 / Published online: 22 May 2018

Abstract We describe the Inspire system which participated in the first competition on inductive logic programming (ILP). Inspire is based on answer set programming (ASP). The distinguishing feature of Inspire is an ASP encoding for hypothesis space generation: given a set of facts representing the mode bias, and a set of cost configuration parameters, each answer set of this encoding represents a single rule that is considered for finding a hypothesis that entails the given examples. Compared with state-of-the-art methods that use the length of the rule body as a metric for rule complexity, our approach permits a much more fine-grained specification of the shape of hypothesis candidate rules. The Inspire system iteratively increases the rule cost limit and thereby increases the search space until it finds a suitable hypothesis. The system searches for a hypothesis that entails a single example at a time, utilizing an ASP encoding derived from the encoding used in XHAIL. We perform experiments with the development and test set of the ILP competition. For comparison we also adapted the ILASP system to process competition instances. Experimental results show that the cost parameters for the hypothesis search space are an important factor for finding hypotheses to competition instances within tight resource bounds.

Keywords Inductive logic programming · Answer set programming · Hypothesis generation · Rule complexity · Best-effort

Editors: James Cussens and Alessandra Russo.

✉ Peter Schüller
ps@kr.tuwien.ac.at
http://www.peterschueller.com

Mishal Benz
mishal.benz@kit.edu

1 Institut für Logic and Computation, Technische Universität Wien, Vienna, Austria

2 Faculty of Engineering, Marmara University, Istanbul, Turkey

3 Karlsruhe Institute of Technology, Karlsruhe, Germany

4 Faculty of Engineering and Natural Science, Sabanci University, Istanbul, Turkey

1 Introduction

Inductive Logic Programming (ILP) (Muggleton et al. 2015) combines several desirable properties of Machine Learning and Logic Programming: logical rules are used to formulate *background knowledge*, and *examples*, which are reasoning inputs paired with desired or undesired reasoning outcomes, are used to learn a *hypothesis*. A hypothesis is an interpretable set of logical rules which entails the examples with respect to the background knowledge. Examples can be noisy, sometimes not all examples can be satisfied, and usually there are several possible hypotheses.

The inaugural competition on Inductive Logic Programming (Law et al. 2016b) featured a family of ILP tasks about agents that are moving in a grid world. Each instance required to find a hypothesis that represents the rules for valid moves of the agent. Some instances required predicate invention, i.e., finding auxiliary predicates that represent intermediate concepts. For example the 'Unlocked' instance required the ILP system to find rules for representing that 'the agent may move to an adjacent cell so long as it is unlocked at that time. A cell is unlocked if it was not locked at the start, or if the agent has already visited the key for that cell.' The competition was open to entries for systems based on Prolog (Clocksin and Mellish 2003) and for systems based on answer set programming (ASP) (Lifschitz 2008; Brewka et al. 2011; Gebser et al. 2012a) and featured a non-probabilistic and a probabilistic track.

In this paper we describe the INSPIRE system which is based on ASP and was the winner of the non-probabilistic competition track, but it was the only entry to that track. The competition was challenging for three main reasons.

(C1) In each instance the examples which were traces of agent movements used overlapping time ranges and the background knowledge contained time comparisons over all earlier time points. Therefore, the wide-spread approach of shifting the time parameter to represent each examples in a distinct part of the Herbrand Base was not possible.[1]
(C2) Computational resources were limited to 30 sec and 2 GB, which is not much for the intractable ILP task.
(C3) Negative example information was given implicitly, i.e., agent movements that were not explicitly given as valid had to be considered invalid for learning.

Also, at the time of the competition, there were no published systems that supported the competition format without the need for significant adaptations.

The INSPIRE system aims to provide a best-effort solution under these conditions. The central novel aspect of our approach is that we generate the hypothesis search space using an ASP encoding that permits a fine-grained cost configuration. We use ASP for all nontrivial computational tasks as shown in the block diagram of our system in Fig. 1.

The idea to iteratively extend the hypothesis search space (in short *hypothesis space*) is present in several existing systems. Our approach of fine-grained cost-based hypothesis generation enables a detailed configuration of rule cost parameters, for example to configure cost for the number of negative body atoms, for variables that are bound only once in the rule body, for invented predicates that are used in the rule body, for the variables that are bound only in the rule head, and for several further rule properties. This provides more control and a

[1] To illustrate this, consider the rule "`visited(C, T) :- agent_at(C, T2), time(T), T >= T2.`" which is part of the background knowledge of instance 17 of the competition. If we represent multiple sequences of `agent_at`$(\cdot,\cdot)$ by allocating time points $0 \ldots 199$ for the first and time points $200 \ldots 399$ for the second agent, then the truth values of atoms of form `visited`$(\cdot,\cdot)$ for the second agent will be influenced by the truth values of atoms of form `agent_at`$(\cdot,\cdot)$ of the first agent.

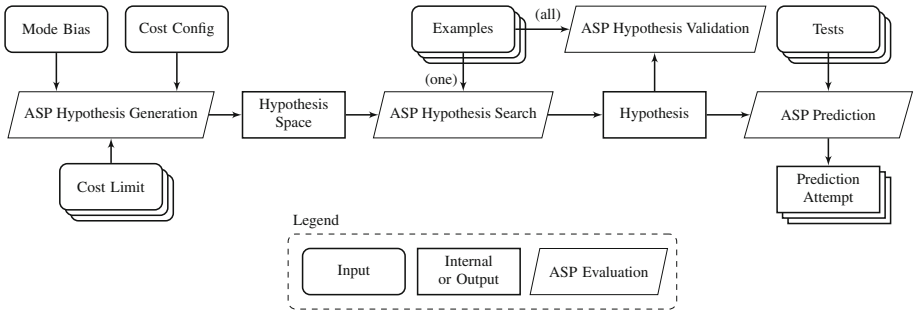

Fig. 1 Data flow of the INSPIRE system, showing inputs, outputs, and ASP evaluations. Background Knowledge is implicitly used in all ASP evaluations except in hypothesis generation

more realistic search space than the common approach of limiting the rule complexity which is measured by counting the number of body literals of a rule. Our approach can be integrated with all ILP systems that first generate a hypothesis space from the mode bias and afterwards search for a hypothesis within that hypothesis space.

According to official competition result, our system predicted 46% of test cases correctly. We performed empirical experiments to investigate reasons for this low accuracy. Increasing the time budget to 10 min increases accuracy on test instances by 18%. We identify learning from a single example at a time as a major reason for wrong predictions. This limitation is due to our hypothesis optimization method which is derived from the one of XHAIL and cannot represent multiple examples that share ground atoms, i.e., it cannot deal with challenge (C1). In a general setting, our fine-grained hypothesis search space is compatible with learning from multiple examples.

We make the following contributions.

- We describe an ASP encoding for generating the hypothesis search space. The encoding permits to attach costs to various aspects of rule candidates. This way the search space exploration can be controlled in a fine-grained way by incrementing a cost limit, and preferences for the shape of rule candidates can be configured easily.
- We give an algorithm that uses this encoding to generate the hypothesis space and learns hypotheses from a single example at a time using a simplification of the XHAIL (Ray 2009) ASP encoding. Each hypothesis is validated on all examples and if the validation score increased since the last validation, a prediction attempt is made, followed by hypothesis learning on the next example. The algorithm is specific to the competition and mainly designed to deal with challenge (C2), i.e., obtaining a reasonable score within tight resource bounds. The algorithm is based on the observation that a single competition example often contained enough structure to learn the full hypothesis.
- We experimentally compare different cost configurations of the INSPIRE system, and we compare our system with the ILASP system (Law et al. 2014). (For that we created a wrapper to adapt ILASP to the competition format and to perform predictions.) Our evaluations show that INSPIRE consistently outperforms ILASP, that there are significant score differences among INSPIRE cost configurations, and that learning from single examples is not sufficient for all competition instances.

In Sect. 2 we provide preliminaries of ASP and ILP and we describe the ILP format. In Sect. 3 we introduce our hypothesis space generation approach, comprising several ASP modules. Section 4 describes the INSPIRE system's algorithm. The empirical evaluation is reported in Sect. 5. We discuss related work in Sect. 7 and conclude in Sect. 8.

2 Preliminaries

2.1 Answer set programming

ASP is a logic programming paradigm which is suitable for knowledge representation and finding solutions for computationally (NP-)hard problems (Gelfond and Lifschitz 1988; Lifschitz 2008; Brewka et al. 2011). We next give preliminaries of ASP programs with uninterpreted function symbols, aggregates and choices. For a more elaborate description we refer to the ASP-Core-2 standard (Calimeri et al. 2012) and to books about ASP (Baral 2004; Gelfond and Kahl 2014; Gebser et al. 2012a).

Syntax. Let $\mathscr{C}$ and $\mathscr{V}$ be mutually disjoint sets of *constants* and *variables*, which we denote with first letter in lower case and upper case, respectively. Constants are used for constant terms, predicate names, and names for uninterpreted functions. The set of *terms* $\mathscr{T}$ is recursively defined: $\mathscr{T}$ is the smallest set containing $\mathbb{N} \cup \mathscr{C} \cup \mathscr{V}$ as well as tuples of form $(t_1, \ldots, t_n)$ and uninterpreted function terms of form $\mathtt{f}(t_1, \ldots, t_n)$ where $\mathtt{f} \in \mathscr{C}$ and $t_1, \ldots, t_n \in \mathscr{T}$. An *ordinary atom* is of the form $\mathtt{p}(t_1, \ldots, t_n)$, where $\mathtt{p} \in \mathscr{C}$, $t_1, \ldots, t_n \in \mathscr{T}$, and $n \geq 0$ is the *arity* of the atom. An *aggregate atom* is of the form $\mathtt{X} = \#agg\{t : b_1, \ldots, b_k\}$ with variable $\mathtt{X} \in \mathscr{V}$, aggregation function $\#agg \in \{\mathtt{\#sum}, \mathtt{\#count}\}$, with $1 < k$, $t \in \mathscr{T}$ and $b_1, \ldots, b_k$ a sequence of atoms. A term or atom is *ground* if it contains no sub-terms that are variables. A *rule* r is of the form α `:-` $\beta_1, \ldots, \beta_n, \mathtt{not}\ \beta_{n+1}, \ldots, \mathtt{not}\ \beta_m$, where $m \geq 0$, α is an ordinary atom, $\beta_j, 0 \leq j \leq m$ is an atom, and we let $B(r) = \{\beta_1, \ldots, \beta_n, \mathtt{not}\ \beta_{n+1}, \ldots, \mathtt{not}\ \beta_m\}$ and $H(r) = \{\alpha\}$. A *program* is a finite set P of rules. A rule r is a *fact* if $m = 0$.

Semantics. Semantics of an ASP program P is defined using its Herbrand Base HB_P and its ground instantiation $grnd(P)$. Given an interpretation $I \subseteq HB_P$ and an atom $a \in HB_P$, I models a, formally $I \models a$, iff $a \in I$ and I models a literal $\mathtt{not}\ a$, formally $I \models \mathtt{not}\ a$, iff $a \notin I$. An aggregate literal in the body of a rule accumulates truth values from a set of atoms, for example $I \models \mathtt{N = \#count\{A : p(A)\}}$ iff the extension of predicate `p` in I, i.e., the set of true atoms of form $\mathtt{p}(\cdot)$, has size N. An interpretation $I \subseteq HB_P$ models a rule r if $I \models B(r)$ or $I \not\models H(r)$, and I models a set of literals if I models all literals. The FLP-reduct (Faber et al. 2011) fP^I reduces a program P using an answer set candidate I: $fP^I = \{r \in grnd(P) \mid I \models B(r)\}$. Finally I is an answer set of P, denoted $I \in \mathbf{AS}(P)$, iff I is a minimal model of fP^I.

Syntactic Sugar. Anonymous variables of form "_" are replaced by new variable symbols. Choice constructions can occur instead of rule heads, they generate a set of candidate solutions if the rule body is satisfied; e.g., `1 { p(a) ; p(b) } 2` in the rule head generates all solution candidates where at least 1 and at most 2 atoms of the set $\{\mathtt{p(a)}, \mathtt{p(b)}\}$ are true (bounds can be omitted). If a term is given as $X..Y$, where $X, Y \in \mathbb{N}$, then the rule containing the term is instantiated with all values from $\{v \in \mathbb{N} \mid X \leq v \leq Y\}$. A *constraint* is a rule r without a head atom, and a constraint eliminates answer sets I where $I \models B(r)$. A constraint can be rewritten into a rule `f :- not f, B(r)`, where f is an atom that does not occur elsewhere in the program.

ASP supports optimization by means of weak constraints which incur a cost instead of eliminating an answer set. We denote by $\mathbf{AS}^{opt,1}(P)$ the first optimal answer set of program P. Note, that the hypothesis space generation encodings, which are the main contribution of this paper, do not require weak constraints because they explicitly represent costs.

2.2 Inductive logic programming

ILP (Muggleton and Raedt 1994; Muggleton et al. 2012) is a combination of Machine Learning with logical knowledge representation. Key advantages of ILP are the generation of compact models that can be interpreted by humans, and the possibility to learn from a small amount of examples. A classical ILP system takes as input a set of examples E, a set B of background knowledge rules, and a set of mode declarations M, also called the mode bias. An ILP system is expected to produce a set of rules H called the hypothesis which entails E with respect to B in the underlying logic programming formalism. The search for H with respect to E and B is restricted by M, which defines a language that limits the shape of rules that can occur in the hypothesis.

Traditional ILP (Muggleton et al. 2012) searches for sets of Prolog rules that entail a given set of positive examples and do not entail a given set of negative examples using SLD resolution and a given set of background theory rules. Brave Induction (Sakama and Inoue 2009) requires that each example is entailed in at least one answer set, while cautious induction requires all examples to be entailed in all answer sets. ILP for ASP was introduced by Otero (2001) and searches for a set of ASP rules that entails each given example (consisting of a positive and negative part) in at least one answer set. ASP hypotheses represent knowledge in a more declarative way than Prolog, i.e., without relying on the SLD(NF) algorithm.

Example 1 Consider the following example ILP instance (M, E, B) (Ray 2009).

$$M = \left\{ \begin{array}{l} \texttt{\#modeh flies(+bird).} \\ \texttt{\#modeb penguin(+bird).} \\ \texttt{\#modeb not penguin(+bird).} \end{array} \right\}$$

$$E = \left\{ \begin{array}{l} \texttt{\#example flies(a).} \\ \texttt{\#example flies(b).} \\ \texttt{\#example flies(c).} \\ \texttt{\#example not flies(d).} \end{array} \right\}$$

$$B = \left\{ \begin{array}{l} \texttt{bird(X) :- penguin(X).} \\ \texttt{bird(a).} \\ \texttt{bird(b).} \\ \texttt{bird(c).} \\ \texttt{penguin(d).} \end{array} \right\}$$

Based on the above, an ILP system would ideally find the following hypothesis.

$$H = \left\{ \texttt{flies(X) :- bird(X), not penguin(X).} \right\}$$

Note that, in this example, the program $B \cup H$ has a single answer set that entails E.

There are also ASP-based ILP systems [for example ILASP (Law et al. 2014)] where a hypothesis must entail each positive examples in some answer set, and no negative example in any answer set. With that, ILP can be used to learn, e.g., the rules of Sudoku: given a background theory that generates all answer sets of 9-by-9 grids containing digits 1 through 9, positive examples of partial Sudoku solutions, and negative examples of partial invalid Sudoku solutions, ILP methods can learn which rules define valid Sudoku solutions. More recently, ASP-based ILP has been extended to inductive learning of preference specifications in the form of weak ASP constraints (Law et al. 2015).

2.3 First inductive logic programming competition

The first international ILP competition, held together with the 26th International Conference on Inductive Logic Programming, aimed to "test the accuracy, scalability, and versatility [of participating ILP systems]" (Law et al. 2016b). The competition comprised a probabilistic and a non-probabilistic track. We consider only the non-probabilistic track here.

The initial datasets consisted of 8 example problems in each track, intended to help entrants build their systems. The datasets used for scoring systems in the competition were completely new and unseen. All runs were made on an Ubuntu 16.04 virtual machine with a 2 GHz dual core processor and resource limits of 2 GB RAM and 30 sec time.

Instances were in the domain of agents moving in a grid world where only some movements were possible. Each instance consists of a background knowledge, a language bias, a set of examples, and a set of test traces. A trace is a set of agent positions at certain time positions. An example contains a trace and a set of valid moves. The ILP system had to learn the rules for possible moves from examples, and then predict for each test trace whether the agent made only valid moves. These predictions were used to produce the final score.

Example 2 In Instance 5, called *Gaps in the floor*, the agent can always move sideways, but can only move up or down in special 'gap' cells which have no floor and no ceiling.

In Instance 11, called *non-OPL transitive links*, the agent may go to any adjacent cell or use given links between cells to teleport. It can also use a chain of links in one go. In this problem, the agent has to learn the (transitive) concept 'linked'.

Each input instance is structured in sections using the following statements (see Fig. 2 for an Example):

- `#background` marks the beginning of the background knowledge.
- `#target_predicate` indicates the predicate which should be defined by the hypothesis, similar to the *modeh* mode declaration in standard language biases.
- `#relevant_predicates` indicates the predicates from the background knowledge that can be used to define the hypothesis, similar to *modeb* mode declarations in standard language biases.
- `#Example(X)` shows the start of an example with identifier `X`. Subsection `#trace` contains the path taken by the agent, and subsection `#valid_moves` gives the complete set of valid moves the agent could take for each time step.
- `#Test(X)` contains a test trace with identifier `X`. Subsection `#trace` contains the path taken by the agent.

Predicates in the language bias contained only variable types as arguments, never constants.

An ILP system in the competition is supposed to learn a hypothesis based on the given examples (traces and valid moves) and the background knowledge, and then predict the valid moves of the given test traces using the learned hypothesis and the background knowledge. The system output consists of answer attempts, which start with `#attempt`, followed by a sequence of lines `VALID(`X`)` or `INVALID(`X`)`, predicting validity of agent movements in each test trace X. Multiple answer attempts are accepted, but only the last one is scored.

```
#background
cell((0,0)). cell((0,1)). cell((1,0)). cell((1,1)).
time(0..100). gap((0,0)). link((0,0),(0,1)).
h_adjacent((X1,Y),(X2,Y)) :- cell((X1,Y)), cell((X2,Y)), X2 = X1 + 1.
v_adjacent((X,Y1),(X,Y2)) :- cell((X,Y1)), cell((X,Y2)), Y2 = Y1 + 1.
v_adjacent(F,G) :- v_adjacent(G,F).  h_adjacent(F,G) :- h_adjacent(G,F).

#target_predicate
valid_move(cell,time)
#relevant_predicates
gap(cell) agent_at(cell,time) h_adjacent(cell,cell) v_adjacent(cell,cell)

#Example(0)
#trace
agent_at((0,0),0).  agent_at((0,1),1).
#valid_moves
valid_move((0,1),0) valid_move((1,0),0) valid_move((1,1),1)

#Test(0)
#trace
agent_at((0,0),0).  agent_at((0,1),1).
```

Fig. 2 Part of Instance 6 from the development set of the competition (Law et al. 2016b)

Example 3 A simplification of Instance 6 from the competition is given in Fig. 2.

3 Declarative hypothesis generation

A central part of our ILP approach is, that we use an ASP encoding to generate the hypothesis space which is the set of rules that is considered to be part of a hypothesis. Concretely, we represent the mode bias of the instance as facts, add them to an answer set encoding which we describe in the following, moreover we add a fact that configures the cost limit for rules in the hypothesis space. Each answer set represents a single rule of the hypothesis space. While this hypothesis space does not permit to find a hypothesis, we increment the cost limit to enlarge the hypothesis space until we find a solution.

We use the following representation for predicate (schemas) $P(t_1, \ldots, t_N)$ with predicate name P, arity N, and argument types $t_1, \ldots, t_N$:

- `pred`(I, P, N) represents predicate P with arity N, where I is a unique identifier for this predicate and arity; and
- `arg`(I, j, t_j) represents the type t_j of argument position j of predicate I.

Example 4 The predicate `valid_move(cell,time)`, which was the target predicate in many instances of the competition, is represented by the following atoms, where `p1` is the unique predicate identifier.

```
pred(p1,valid_move,2).
arg(p1,1,cell).
arg(p1,2,time).
```

3.1 Input representation

Hypothesis space generation is based on a target predicate and relevant predicates of the instance at hand. We represent this input in atoms of the following form, using above schema:

- `tpred`(I, P, N) for the target predicate;
- `targ`(I, J, T) for arguments of the target predicate;
- `rpred`(I, P, N) for relevant predicates;
- `rarg`(I, J, T) for arguments of relevant predicates; and
- `type_id`(T, *ID*) for all types T used in the target predicate and in relevant predicates, where *ID* is the *type identifier*, a unique integer associated with T. The set of all type identifiers must form a zero-based continuous sequence.

Example 5 The mode bias that is given in Fig. 2 is represented as follows. The target predicate `valid_move(cell,time)` is represented by the following facts, is `t1` is an identifier for the predicate, and `1` and `2` in `targ` are argument positions of the predicate.

```
1 tpred(t1,valid_move,2). targ(t1,1,cell). targ(t1,2,time).
```

The relevant predicates `gap(cell)`, `agent_at(cell,time)`, `h_adjacent(cell,cell)`, and `v_adjacent(cell,cell)`, are represented by the following facts, where `r1`, …, `r4` are the respective predicate identifiers.

```
2 rpred(r1,gap,1).         rarg(r1,1,cell).
3 rpred(r2,agent_at,2).    rarg(r2,1,cell). rarg(r2,2,time).
4 rpred(r3,h_adjacent,2). rarg(r3,1,cell). rarg(r3,2,cell).
5 rpred(r4,v_adjacent,2). rarg(r4,1,cell). rarg(r4,2,cell).
```

Finally, the following facts define type identifiers `0` and `1` for `cell` and `time`, respectively.

```
6 type_id(cell,0).
7 type_id(time,1).
```

3.2 Output representation

We represent a single rule per answer set during hypothesis space generation.

A rule is represented in atoms of the following form:

- `use_var_type`(V, T) represents that the rule uses variable V with type T. V is a term of form `v`(*Idx*) denoting variable with index *Idx* and T is a type as provided in input atoms as first argument of predicate `type_id` for relevant predicates.
- `use_head_pred`(*Id*, *Pred*, *A*) represents that the rule head is an atom with predicate identifier *Id*, predicate *Pred*, and arity *A*.
- `use_body_pred`(*Id*, *Pred*, *Pol*, *A*) represents that the rule body contains a literal with literal identifier *Id*, predicate *Pred*, of polarity *Pol*, and arity *A*. Importantly, if a predicate is used in multiple body literals, *Id* is different for each literal; *Id* is also used in `bind_bvar` (see below) for binding variables to argument positions of particular literals.
- `bind_hvar`(J, V) represents that the argument position J in the rule head contains variable V, where V is a term of form $v(Idx)$.

– `bind_bvar`(*Id*, *Pol*, *J*, *V*) represents that the rule body literal with identifier *Id* and polarity *Pol* contains in its argument position *J* the variable *V*. (*Id* refers to a unique body literal as represented in the argument *Id* of `use_body_pred`, see above.)

Example 6 The hypothesis candidate

```
valid_move(V5,V10) :- cell(V5), time(V10), not agent_at(V5,V10).
```

is represented in an answer set by the following atoms.

```
use_var_type(v(5),cell)
use_var_type(v(10),time)
use_head_pred(t1,valid_move,2)
bind_hvar(1,v(5))
bind_hvar(2,v(10))
use_body_pred(id_idx(r2,1),agent_at,neg,2)
bind_bvar(id_idx(r2,1),neg,1,v(5))
bind_bvar(id_idx(r2,1),neg,2,v(10))
```

As in Example 5, `t1` represents target predicate `valid_move(cell,time)` and `r2` represents the relevant predicate `agent_at(cell,time)`.

Line 1 represents that variable `V5` has type `cell` and line 2 represents that variable `V10` has type `time`. Line 3 represents that the rule head contains target predicate `valid_move`, and lines 4 and 5 represent that the first and second argument positions of the head atom are bound to the variables `V5` and `V10`, respectively. Lines 6–8 represent the body literal `notagent_at`(·,·): line 6 represents that the body contains a negated literal with predicate `agent_at`, and this literal has the unique identifier `id_idx(r2,1)`; lines 7–8 represent that the first and second argument positions of this literal are bound to variables `V5` and `V10`, respectively.

The body literals `cell(V5)` and `time(V10)` are not explicitly represented, they are implicit from `use_var_type`.

3.3 Cost configuration

For fine-grained control over the shape of rules in the hypothesis space, we define several cost components on rules. Intuitively, rules with lower overall cost will be considered in the search space earlier than rules with higher cost. All rules below a certain cost are used simultaneously for finding a hypothesis that entails a given example (see also Sect. 4 and Algorithm 1).

Hard Limits. For ensuring decidability, it is necessary to impose hard limits on the overall size of the hypothesis space. We use the following hard restrictions on rules in the hypothesis search space. Configuration parameters are written in bold and default values are given in brackets.

- **maxvars** (4) specifies the maximum number of variables per type. This limits how many variables of a single type can occur simultaneously in one rule.
- **maxuseppred** (2) specifies the maximum occurrence of a single predicate as a positive body literal. This limits how often we can use the same predicate in the positive rule body. For example, for obtaining a transitive closure in the hypothesis space, this value needs to be at least 2, and **maxvars** needs to be at least 3.
- **maxusenpred** (2) specifies the maximum occurrence of a single predicate as a negative body literal.

- **maxliterals** (4) specifies the maximum number of overall literals in a rule. This imposes a hard limit on the size of hypothesis rule bodies.
- **maxinventpred** (1) specifies the maximum number of predicates to be invented.
- **inv_minarity** (2) specifies the minimum arity of invented predicates.
- **inv_maxarity** (2) specifies the maximum arity of invented predicates.

These hard limits make the hypothesis search space finite by limiting the usage of the mode bias, therefore their values must be chosen with care. Moreover, these limits determine the size of the instantiation of the ASP encoding, which influences the efficiency of enumerating answer sets. In Sect. 4.1 we discuss soundness and completeness of the INSPIRE system with respect to this finite search space.

Fine-grained cost configuration. For configuring fine-grained hypothesis generation, we provide the following cost parameters for various aspects of rules in the hypothesis search space. (Defaults are again given in brackets.)

– **free_vars** (2) specifies the number of variables that do not incur cost.
– **cost_vars** (1) specifies the cost for each variable beyond **free_vars**.
– **cost_type_usedmorethantwice** (2) specifies the cost for each usage of a type beyond the second usage. For example, this cost is incurred if we use three variables of type `time` in one rule.
– **cost_posbodyliteral** (1) specifies the cost for each positive body literal.
– **cost_negbodyliteral** (2) specifies the cost for each negative body literal. The default value is higher than the one for **cost_posbodyliteral**, because usually programs have more positive body literals than negative body literals.
– **cost_pred_multi** (2) specifies the cost for repeated usage of a predicate in the rule body. The cost is incurred for each usage after the first usage. Usage is counted separately for positive and negative literals.
– **cost_varonlyhead** (5) specifies the cost for each variable that is used only in the head. Rules with such variables are still safe because each variable has a type. It is possible, but rare, that such rules are useful, so they obtain high cost.
– **cost_varonlyoncebody** (5) specifies the cost for each variable that is not used in the head and used only once in the body of the rule. Such variables are like anonymous variables and project away the argument where they occur. We expect such cases to be rare so we incur a high default cost.
– **cost_var_boundmorethantwice** (2) specifies the cost for each variable that occurs in more than two literals.
– **cost_reflexive** (5) specifies the cost for each binary atom in the rule body with the same variable in both arguments.
– **cost_inv** (2) specifies the cost for inventing any kind of predicate. This cost is used to adjust above which cost predicate invention is performed, independent from the cost of inventing each predicate.
– **cost_inv_pred** (2) specifies the cost for each invented predicate.
– **cost_inv_headbody** (3) specifies the cost for using the same invented predicate both in the head and in the body of the same rule.
– **cost_inv_bodymulti** (5) specifies the cost for using multiple invented predicates in the rule body.
– **cost_inv_headbodyorder** (5) specifies the cost for using an invented predicates in the head and a different invented predicate in the body of a rule, in a way that the head predicate is lexically greater than the body predicate. This incurs higher cost to programs with cycles over invented predicates, and less cost to those that have no such cycles. In

(i) Hypothesis candidates with cost 1.

```
valid_move(V5,V10) :- agent_at(V5,V10),cell(V5),time(V10).
```

(ii) Hypothesis candidates with cost 2.

```
valid_move(V5,V10) :- not agent_at(V5,V10),cell(V5),time(V10).
```

(iii) Hypothesis candidates with cost 3.

```
valid_move(V5,V10) :- agent_at(V6,V10),h_adjacent(V5,V6), ... .
valid_move(V5,V10) :- agent_at(V6,V10),h_adjacent(V6,V5), ... .
valid_move(V5,V10) :- agent_at(V6,V10),v_adjacent(V5,V6), ... .
valid_move(V5,V10) :- agent_at(V6,V10),v_adjacent(V6,V5), ... .
```

(iv) Hypothesis candidates with cost 4.

```
valid_move(V5,V10) :- agent_at(V5,V10),gap(V5),cell(V5),time(V10).
valid_move(V5,V10) :- agent_at(V6,V10),not h_adjacent(V5,V6), ... .
valid_move(V5,V10) :- agent_at(V6,V10),not h_adjacent(V6,V5), ... .
valid_move(V5,V10) :- agent_at(V6,V10),not v_adjacent(V5,V6), ... .
valid_move(V5,V10) :- agent_at(V6,V10),not v_adjacent(V6,V5), ... .
valid_move(V5,V10) :- h_adjacent(V5,V6),not agent_at(V6,V10), ... .
valid_move(V5,V10) :- h_adjacent(V6,V5),not agent_at(V6,V10), ... .
valid_move(V5,V10) :- v_adjacent(V5,V6),not agent_at(V6,V10), ... .
valid_move(V5,V10) :- v_adjacent(V6,V5),not agent_at(V6,V10), ... .
```

(v) Hypothesis candidates with cost 5.

```
valid_move(V5,V10) :- agent_at(V5,V10),not gap(V5),cell(V5),time(V10).
valid_move(V5,V10) :- gap(V5),not agent_at(V5,V10),cell(V5),time(V10).
valid_move(V5,V10) :- agent_at(V6,V10),gap(V5),not gap(V6), ... .
valid_move(V5,V10) :- agent_at(V6,V10),gap(V6),not gap(V5), ... .
valid_move(V5,V10) :- not agent_at(V6,V10),not h_adjacent(V5,V6), ... .
valid_move(V5,V10) :- not agent_at(V6,V10),not h_adjacent(V6,V5), ... .
valid_move(V5,V10) :- not agent_at(V6,V10),not v_adjacent(V5,V6), ... .
valid_move(V5,V10) :- not agent_at(V6,V10),not v_adjacent(V6,V5), ... .
valid_move(V5,V10) :- ip_1_2(V5,V10),cell(V5),time(V10).
ip_1_2(V5,V10) :- agent_at(V5,V10),cell(V5),time(V10).
ip_1_2(V5,V6) :- h_adjacent(V5,V6),cell(V5),cell(V6).
ip_1_2(V5,V6) :- h_adjacent(V6,V5),cell(V5),cell(V6).
ip_1_2(V5,V6) :- v_adjacent(V5,V6),cell(V5),cell(V6).
ip_1_2(V5,V6) :- v_adjacent(V6,V5),cell(V5),cell(V6).
```

Fig. 3 Hypothesis candidates and costs under default cost parameters for Instance 6, see Fig. 2. For space reasons, we abbreviate "`cell(V5),cell(V6),time(V10)`" as "`...`"

particular this allows for an early consideration of hypotheses that use invented predicates in a "stratified" way such that they have no chance to introduce nondeterminism (by means of even loops over invented predicates).

The above costs can independently be adjusted and influence the performance of our approach. Costs determine the stage of the ILP search at which a certain rule will be used as a candidate for the hypothesis.

If we expect certain rules to be more useful for finding a hypothesis in a concrete application, they should be configured to have lower cost than other rules.

Example 7 The hypothesis candidate shown in Example 6 has a cost of 2 using the default cost settings, because of one cost component from **cost_negbodyliteral**.

Figure 3 shows all rules of cost 1–5 for the instance given in Fig. 2 using default cost parameters.

In (i), only the cost parameter **cost_posbodyliteral** $= 1$ is effective. Similarly, in (ii), only **cost_negbodyliteral** $= 2$ has an effect.

In (iii), for a total cost of 3, each candidate obtains cost **cost_vars** $= 1$ for using three variables (two variables incur no cost because **free_vars** $= 2$) and each candidate obtains cost 2 for two positive body literals.

In (iv), for a total cost of 4, the first rule obtains cost **cost_var_boundmorethantwice** $= 2$ for `V5` which is used three times in rule head and rule body, moreover cost 2 for two positive body literals. Each of the remaining eight rules in (iv) obtains **cost_vars** $= 1$ for using three variables, cost 1 for a positive body literal, and cost 2 for a negative body literal.

In (v), for a total cost of 5, the hypothesis candidates in lines 1–2 obtain cost **cost_var_boundmorethantwice** $= 2$ for using `V5` three times and cost for one positive and one negative body literal. Candidates in lines 3–4 obtain cost **cost_vars** $= 1$ for using three variables, cost 2 for two positive body literals, and cost 2 for one negative body literal. Candidates in lines 5–8 obtain cost **cost_vars** $= 1$ for using three variables and cost 2 for two negative body literals. Candidates in lines 9–14 obtain cost **cost_inv** $= 2$ for inventing predicates, **cost_inv_pred** $= 2$ for the first invented predicate `ip_1_2`, and cost 1 for a positive body literal.

Note that the shown rules are the actual internal system representation where redundant candidates (with renamed variables) have been eliminated (see Sect. 3.6).

The fine-grained nature of this cost configuration becomes apparent when considering the "classical" cost notion of rule body length: all rules in (i) and (ii) and rule in lines 9–14 of (v) have bodies of length 1, rules in lines 2 and 3 of (v) have bodies of length 3, and all remaining rules shown in Fig. 3 have bodies of length 2.

Different from hard limits, adjusting cost parameters has no influence on the possibility to find a hypothesis. Instead, these parameters intuitively control search heuristics: they influence in which order hypotheses are considered and therefore can speed up or slow down hypothesis search.

3.4 Main encoding

The main encoding for generating the hypothesis space is given in Fig. 4. Hard limits and cost parameters are added to this encoding as constant definitions.

We define distinct typed variables in atoms of form `var_type`(v(*Index*),*T*) in line 1. Such an atom represents, that the variable of form v(*Index*) has type *T*, where *Index* is a running index over all variables. This defines **maxvars** variables of each type. Note that we use v(*Idx*) to enable a later extension of our encodings with constant strings as arguments in the mode bias. Constant strings were not required for the ILP competition.

Head predicates are represented as `hpred` and `harg`, and body predicates are represented as `bpred` and `barg` in lines 2–5. These are defined from target predicate and relevant predicate, respectively. We define further head and body predicates in the encoding for invented predicates (see Sect. 3.7).

We guess how many variables (up to **maxvars**) are in the rule in the current answer set candidate (line 6). We guess which concrete variables (including their type) are in the rule (line 7). Atoms of form `use_var_type`(*V*, *T*) represent that variable *V* has type *T*.

Lines 8–11 define unique placeholders for all potentially existing literals in the rule body according to the given hard limits. Such placeholders are represented in atoms of form `body_pred(ID,Pred,Pol,A)` where *ID* is a unique term built for that predicate from its predicate identifier *Id* and a running index *Idx*, *Pred* is the predicate name, *Pol* the polarity, and *A* the arity of the predicate.

A subset of these placeholders is guessed as a body literal in lines 12–13, up to a maximum of **maxliterals** literals. A guess in line 14 determines the head predicate. Up to this point,

```
1  var_type(v(ID*maxvars+Idx),T) :- type_id(T,ID), Idx=0..(maxvars-1).

2  hpred(I,P,N) :- tpred(I,P,N).
3  harg(I,J,T) :- targ(I,J,T).
4  bpred(I,P,N) :- rpred(I,P,N).
5  barg(I,J,T) :- rarg(I,J,T).

6  1 { varcount(1..maxvars) } 1.
7  VC { use_var_type(V,T): var_type(V,T) } VC :- varcount(VC).

8  body_pred(id_idx(Id,Idx),Pred,pos,Arity) :-
9    bpred(Id,Pred,Arity), Idx = 1..maxuseppred.
10 body_pred(id_idx(Id,Idx),Pred,neg,Arity) :-
11   bpred(Id,Pred,Arity), Idx = 1..maxusenpred.

12 1 { use_body_pred(id_idx(Id,Idx),Pred,Polarity,Arity)
13     : body_pred(id_idx(Id,Idx),Pred,Polarity,Arity) } maxliterals.
14 1 { use_head_pred(Id,Pred,NArgs) : hpred(Id,Pred,NArgs) } 1.

15 1 { bind_hvar(Pos, VId) : use_var_type(VId,Type) } 1 :-
16       use_head_pred(PredId,_,_), harg(PredId,Pos,Type).
17 1 { bind_bvar(id_idx(PredId,Idx),Polarity,Pos,VId)
18     : use_var_type(VId,Type) } 1 :-
19       use_body_pred(id_idx(PredId,Idx),_,Polarity,_), barg(PredId,Pos,Type).

20 hbound_var(VId) :- bind_hvar(_,VId).
21 bbound_var(VId) :- bind_bvar(_,_,_,VId).

22 :- use_var_type(VId,_), not hbound_var(VId), not bbound_var(VId).
```

Fig. 4 Main module for ASP hypothesis generation

the encoding represents variables including their type, which variables are going to be used, and which head and body predicates to use as literals.

In lines 15–19, we perform a guess for binding these variables to particular argument positions of head and body predicates. The limits of th ese choice rules require that each position is bound exactly once. Moreover, the conditions within the choice ensure that variables are bound to argument positions of the correct type. In lines 20–21, we represent the set of variables that are bound in the head and the same for the rule body (this separation is used for cost representations). Finally, an answer set where a variable is used but neither bound to the head nor to the body of the rule, is eliminated by the constraint in line 22. We do not forbid 'unsafe rules' (where a variable exists only in the head of a rule) because all variables are typed and therefore each variable occurs in an implicit domain predicate in the rule body.

If we evaluate this program module together with a mode bias given as facts according to Sect. 3.1 and together with constant definitions of hard limits according to Sect. 3.3, we obtain answer sets that represent single rules according to Sect. 3.2 and according to the given mode bias and hard limits.

3.5 Fine-grained cost module

Figure 5 shows the encoding module for representing the cost of a rule according to cost configuration parameters described in Sect. 3.3.

```
1  cost(varcount,dummy,(Count-free_vars)*cost_vars) :-
2    varcount(Count), Count > free_vars.

3  cost(vartype_morethantwice,Type,cost_type_usedmorethantwice*(NUse-2)) :-
4    NUse > 2, NUse = #count { Id : use_var_type(v(Id),Type) }, type_id(Type,_).

5  cost(pbodylit,IdIdx,cost_posbodyliteral) :- use_body_pred(IdIdx,_,pos,_).
6  cost(nbodylit,IdIdx,cost_negbodyliteral) :- use_body_pred(IdIdx,_,neg,_).
7  cost(pmulti,id_idx(P,Idx),cost_pred_multi) :-
8    use_body_pred(id_idx(P,Idx),_,_,_),Idx > 1.

9  cost(varonlyh,V,cost_varonlyhead) :-
10   use_var_type(V,_), hbound_var(V), not bbound_var(V).

11 cost(varonlyonceb,V,cost_varonlyoncebody) :- use_var_type(V,_),
12   not hbound_var(V), 1 = #count { Pred,Pol : bind_bvar(Pred,Pol,_,V) }.

13 cost(varmorethantwice,V,cost_varboundmorethantwice*(N-2)) :-
14   use_var_type(V,_), N > 2,
15   N = #count { h,Pos : bind_hvar(Pos,V) ;
16                b,Pred,Pol,Pos : bind_bvar(Pred,Pol,Pos,V) }.

17 reflexive(id_idx(Pr,Idx),Pol) :-
18   bind_bvar(id_idx(Pr,Idx),Pol,1,V), use_body_pred(id_idx(Pr,Idx),Pred,Pol,2),
19   bind_bvar(id_idx(Pr,Idx),Pol,2,V).
20 cost(reflexive,id_idx_pol(Pr,Idx,Pol),cost_reflexive) :-
21   reflexive(id_idx(Pr,Idx),Pol).

22 totalcost(C) :- C = #sum { Cost,U,V : cost(U,V,Cost) }, C < climit.
23 totalcost(climit) :- climit <= #sum { Cost,U,V : cost(U,V,Cost) }.
24 :- totalcost(C), C >= climit.
```

Fig. 5 Fine-grained cost module for ASP hypothesis generation

Cost atoms of the form `cost`(*Name*, *Data*, *Cost*) represent various costs that add up to the total rule cost. Each cost atom bears a name *Name* used to distinguish different aspects of cost. The argument *Data* specifies different elements of the same rule (e.g., variables, literals) that can contribute cost under that aspect. Finally *Cost* is the actual cost incurred for *Name* and *Data*.

Example 8 For the aspect of using a variable type more than twice, the cost aspect is *Name* = `vartype_morethantwice`, and *Data* contains the variable type for which this cost is incurred. For each variable type, this aspect incurs cost separately, which leads to multiple atoms with different *Data* values.

Lines 1–2 define a cost for the number of distinct variables that are used in the rule, where the first **free_distinct_variables** variables incur no cost. Lines 3–4 define the cost for variable types that are used more than twice (using a type once or twice is free). Lines 5–6 define costs for positive and negative body literals, and lines 7–8 define costs for using a predicate multiple times in the body. Note, that line 7 relies on the property that the body literals of lowest index are used, which is ensured by the redundancy elimination module (see Sect. 3.6, lines 2–3 in Fig. 6). Lines 9–10 define a cost for each variable that occurs only in the head. Lines 11–12 define a cost for variables that occur only once in the body of the rule and not in the rule head (these act as anonymous variables). Lines 13–16 define a cost for variables that occur more than twice in the rule. Lines 17–21 define a cost for the reflexive

```
1  use_var_type(v(Id-1),Type) :- use_var_type(v(Id),Type), var_type(v(Id-1),Type).
2  use_body_pred(id_idx(PredId,Idx-1),Pred,Polarity,Arity) :-
3      use_body_pred(id_idx(PredId,Idx),Pred,Polarity,Arity), Idx > 1.

4  :- bind_hvar(Pos,v(Id)),
5     var_type(v(Id),T), var_type(v(Id-1),T), not hbound_var(v(Id-1)).
6  :- bind_hvar(1,Id1), bind_hvar(2,Id2),
7     var_type(Id1,T), var_type(Id2,T), Id1 > Id2.

8  lit_vsig(Id,Idx,Pol,1,V) :- use_body_pred(id_idx(Id,Idx),_,Pol,1),
9                              bind_bvar(id_idx(Id,Idx),Pol,1,V).
10 lit_vsig(Id,Idx,Pol,2,vv(V1,V2)) :- use_body_pred(id_idx(Id,Idx),_,Pol,2),
11                                     bind_bvar(id_idx(Id,Idx),Pol,1,V1),
12                                     bind_bvar(id_idx(Id,Idx),Pol,2,V2).

13 :- lit_vsig(Id,Idx1,Pol,A,S1), lit_vsig(Id,Idx2,Pol,A,S2), Idx1<Idx2, S1>S2.

14 litequal_upto(Id1,Id2,Pol1,Pol2,0,Arity) :-
15     use_body_pred(Id1,Pred,Pol1,Arity),
16     use_body_pred(Id2,Pred,Pol2,Arity), (Id1, Pol1) < (Id2, Pol2).
17 litequal_upto(Id1,Id2,Pol1,Pol2,Upto+1,Arity) :-
18     litequal_upto(Id1,Id2,Pol1,Pol2,Upto,Arity), Upto < Arity,
19     bind_bvar(Id1,Pol1,Upto+1,VId), bind_bvar(Id2,Pol2,Upto+1,VId).
20 litequal(Id1,Id2) :- litequal_upto(Id1,Id2,_,_,Arity,Arity).
21 :- litequal(Id1,Id2).
```

Fig. 6 Redundancy elimination module for ASP hypothesis generation

usage of a binary predicate, i.e., a cost for literals that contain the same variable in both arguments. Lines 22–23 sum up the total cost if that total is below `climit`, otherwise the total cost is fixed to `climit`.[2] Finally, solutions that reach or exceed a total cost of `climit` are excluded in line 24.

3.6 Redundancy elimination module

The encoding in Fig. 4 creates many solutions that produce the same or a logically equivalent rule. As an example, line 1 defines **maxvars** variables of the same type, and line 7 guesses which of these variables to use. If variables with index 1 and 2 have the same type, there can be two answer sets which represent two rules that are different modulo variable renaming. For example, one of the rules "`foo(V1) :- bar(V1).`" and "`foo(V2) :- bar(V2).`" is redundant. Redundant rules make the hypothesis search slower and do not contribute to the solution, therefore they should be avoided.

Figure 6 is an ASP module that eliminates most redundancies and thereby improves performance without losing any potential hypotheses. Line 1 ensures that if we use variables of a certain type, we use only variables with the lowest index of that type. This is realized by enforcing that variable v(Id-1) is used whenever variable v(Id) is used and under the condition that these variables have the same type. Similarly, lines 2–3 require that those body literals that have the lowest indexes are used. Lines 4–7 canonicalize variables that are used in rule heads: the constraint in lines 4–5 requires that the variables with lowest index are bound in the head, and the constraint in line 6–7 rules out solutions where two variables of

[2] The rule `totalcost(C) :- C = #sum { Cost,U,V : cost(U,V,Cost) }.` would sum up total cost in a single rule, without clamping the value to the maximum interesting value `climit`. However, this naive approach has a significantly larger instantiation than the encoding we use here.

```
1  0 { ipred(ip(Id,A),A) : A = inv_minarity..inv_maxarity } 1 :-
2    Id = 1..maxinventpred.
3  1 { iarg(Id,Pos,T) : type(T) } 1 :- ipred(Id,A), Pos = 1..A.

4  hpred(Id,Id,A) :- ipred(Id,A).
5  harg(I,J,T) :- iarg(I,J,T).
6  bpred(Id,Id,A) :- ipred(Id,A).
7  barg(I,J,T) :- iarg(I,J,T).

8  :- iarg(Id,Pos1,T1), iarg(Id,Pos2,T2), Pos1 < Pos2, T2 < T1.

9  use_ipred(Id) :- ipred(Id,_), use_head_pred(Id,_,_).
10 use_ipred(Id) :- ipred(Id,_), use_body_pred(id_idx(Id,_),_,_,_).
11 :- ipred(Id,_), not use_ipred(Id).

12 cost(inv,dummy,cost_inv) :- ipred(_,_).
13 cost(inv_pred,Id,cost_inv_pred) :- ipred(Id,Arity).
14 cost(inv_headbody,ip(Id,A),cost_inv_headbody) :-
15   use_head_pred(ip(Id,A),_,_), use_body_pred(id_idx(ip(Id,A),Idx),_,_,A).
16 cost(inv_bodymulti,id_idx_idx(ip(Id,A),Idx1,Idx2),cost_inv_bodymulti) :-
17   use_body_pred(id_idx(ip(Id,A),Idx1),_,_,A), Idx1 < Idx2,
18   use_body_pred(id_idx(ip(Id,A),Idx2),_,_,A).
19 cost(inv_headbodyorder,
20      h_b(ip(Id1,A1),id_idx(ip(Id2,Idx2))),
21      cost_inv_headbodyorder) :-
22   use_head_pred(ip(Id1,A1),_,A1), Id1 >= Id2,
23   use_body_pred(id_idx(ip(Id2,Idx2),_),_,_,_).
```

Fig. 7 Predicate invention module for ASP hypothesis generation

the same type are used in the head where the variable with the lower index is used in the second argument of the predicate. This rules out, e.g., a rule with the head `foo(X2,X1)` if `X1` and `X2` have the same type.

For further redundancy elimination we rely on the lexicographic order of terms in ASP. In lines 8–12 we define atoms which represent a *variable signature S* of form `lit_vsig`(I, Idx, Pol, A, S) where I is the predicate identifier, Idx the index, Pol the polarity, A the arity, and S is a composite term containing all variables in the literal for which the signature is defined. Using these signatures, the constraint in line 13 requires that literals with equal predicates and polarities are sorted in the same way as their variable signatures. This would, for example, eliminate a rule with the body "`foo(X2,X3), foo(X1,X2)`" while it would allow to use the logically equivalent body "`foo(X1,X2), foo(X2,X3)`". Finally, in lines 14–20, we represent pairs of literals that have the same predicate and the same arguments (and potentially different polarity), and we exclude solutions that contain a pair of such equal literals using the constraint in line 21.

Note that lines 6–12 of this encoding are suitable only for predicates with arity one or two. This was sufficient for the ILP competition and can be generalized to higher arities.

3.7 Predicate invention module

Figure 7 shows the ASP module which extends the search bias by adding the possibility to invent predicates in the hypothesis search space. This module also includes redundancy elimination aspects that are specific for predicate invention.

Lines 1–2 define a guess over the arity of up to **maxinventpred** invented predicates. Guessing no arity means that the invented predicate is not used. Line 3 guesses for each

invented predicate and for each argument position one argument type. Lines 4–7 connect the invented predicate encoding with the main encoding by defining that invented predicates can be used both as head as well as body predicates in hypothesis rules. Line 8 requires that arguments of invented predicates are sorted lexically in the same way as the IDs of their types are. This reduces redundancy, because it does not matter in practice whether we use an invented predicate as `inv(Type1,Type2)` or as `inv(Type2,Type1)`, as long as it is used in the same way in all rules. Lines 9–11 perform further redundancy elimination by defining which invented predicates are used, and eliminating solutions where we guess the existence of an invented predicate but do not use it. Line 12 defines a cost for predicate invention in general, line 13 defines a cost for each invented predicate, lines 14–15 define extra cost if the invented predicate is used both in the head and in the body of the rule in the answer set, lines 16–18 define extra cost for multiple usages of the same predicate in the rule body, and lines 19–23 define a cost for pairs of distinct invented predicates where one is in the body and the other one in the head of a hypothesis rule: cost is defined if the predicate in the head is lexicographically greater or equal to the predicate in the body. This prevents rules that can make cycles over invented predicates early in the search process, and intuitively prefers hypotheses that are stratified (Apt et al. 1988) with respect to invented predicates.

4 Best-effort learning and prediction

The INSPIRE system performs brave induction of explicitly given positive and implicitly given negative examples, which was sufficient for the competition. The type of induction task that is solved is similar to the task solved by the XHAIL system. Formally, INSPIRE searches for a hypothesis H such that for each given example trace $\langle trace, valid\_moves \rangle$ there is an $I \in \mathbf{AS}(bk \cup trace \cup H)$ with I containing the valid moves specified in *valid_moves* and no other valid moves.

Algorithm 1 shows the main algorithm of the INSPIRE system which is visualized from a conceptual point of view in Fig. 1 (see page 3). The algorithm gets a competition instance (see Sect. 2.3) as input: background knowledge *bk* is a set of ASP rules, the set of examples *e* is of form $\langle trace, label \rangle$ where *trace* is a set of atoms for predicate `agent_at` and *label* is a set of atoms for predicate `valid_move`, the mode bias *m* is given in the form of target predicate and relevant predicates, and finally the set of test traces *tests* is of form $\langle trace, id \rangle$ where *trace* is a set of atoms for predicate `agent_at` and *id* is required for labeling prediction outputs with the correct trace.

Initially, Algorithm 1 sorts examples by length of their trace. The variable *bestquality*, initially zero, stores the number of examples that we can predict correctly with the best hypothesis found so far. The loop in line 4 iterates over the sorted examples, starting with the smallest. For each example, the loop in line 5 iterates over cost limit values from $climit_{min}$ to $climit_{max}$. For the competition, we set $climit_{min} = 4$ and $climit_{max} = 15$, in a general setting we use $climit_{min} = 1$ and $climit_{max} = \infty$. For each value of *climit*, in line 6 we enumerate all answer sets of $P_{hypgen}(m, climit)$ which denotes the ASP encodings for hypothesis generation as described in Figs. 4, 5, 6 and 7. The parameters *m* and *climit* of encoding P_{hypgen} are used as follows: from the mode bias *m*, facts are generated as described in Sect. 3.1, and the value of *climit* is passed to ASP as a constant `climit`. In line 7 each answer set is transformed into a (nonground) rule which yields the hypothesis search space *hspace*.

Given *hspace*, we search for an optimal hypothesis using the current example's *trace* and *label*. A hypothesis $h \subseteq hspace$ must predict the extension of `valid_move` in the

Algorithm 1: INSPIRE- ILP(KB *bk*, Examples *e*, Mode-bias *m*, Test-traces *tests*)

```
1  Sort examples e by length of trace                    // process smaller examples first
2  bestquality := 0
3  besthypothesis := null
4  for ⟨trace, label⟩ ∈ e do                             // process sorted examples one by one
5      for climit = climit_min, ..., climit_max do
6          ha := AS(P_hypgen(m, climit))                 // hypothesis search space generation
7          hspace := rules extracted from answer sets ha as described in Section 3.2
8          P_hs := bk ∪ trace ∪ {P_rule(r) | r ∈ hspace} ∪ P_verify(label)   // build search program
9          h := AS^{opt,1}(P_hs)                         // hypothesis search and optimization
10         if h exists then                              // found a hypothesis for this example
11             quality := |{⟨etrace, elabel⟩ ∈ e | elabel is exactly reproduced in AS(bk ∪ h ∪ etrace)}|
12             if quality > bestquality then
13                 bestquality := quality
14                 besthypothesis := h
15                 Print "#attempt"                      // best-effort output
16                 for ⟨testtrace, testid⟩ ∈ tests do
17                     if all agent movements in AS(bk ∪ h ∪ testtrace) are predicted as valid then
18                         Print "VALID(testid)"
19                     else
20                         Print "INVALID(testid)"
21                 if quality = |e| then                 // h predicts all examples in e correctly
22                     return h
23 return besthypothesis
```

label correctly with respect to background knowledge *bk* and *trace*. Note, that predicate `valid_move` is specific to the competition, but our encodings are flexible with respect to using different or even multiple predicates. A correct hypothesis is optimal if it has lower or equal cost compared with all other correct hypotheses, where cost is the sum of costs of rules used in the hypothesis and cost of a single rule is computed according to Sect. 3.3.

Hypothesis search is done using ASP optimization on a program P_{hs} whose encoding is similar to the Inductive Phase encoding used in XHAIL (Ray 2009, Section 3.3). Briefly, P_{hs} contains the background knowledge, the current example's trace *trace* as facts, a module $P_{verify}(label)$ which eliminates solutions that do not satisfy the example, and a transformed rule $P_{rule}(r)$ for each rule r in the hypothesis space. $P_{verify}(label)$ contains the set {`pos_valid_move(X,Y).` | `valid_move(X,Y)` ∈ *label*} of facts which encode positive example traces; a rule "`covered :-` *label*." which recognizes coverage of positive example parts; and the following rules which recognize the entailment of the example by checking the coverage of positive examples and forbidding the violation of invalid moves:

```
violated :- valid_move(X,Y), not pos_valid_move(X,Y).
good_example :- covered, not violated.
:- not good_example.
```

$P_{rule}(r)$ contains (i) the original rule r with an additional body condition `use(`r`)`; (ii) a guess `{use(`r`)}`. which determines whether rule r is part of the hypothesis; and (iii) a weak constraint that incurs the cost of r if `use(`r`)` is true. In each answer set I of P_{hs}, the set of rules {r | `use(`r`)`} ⊆ *hspace* is a hypothesis that entails the given example ⟨*trace*, *label*⟩. An

optimal answer set of P_{hs} is a hypothesis such that there is no cheaper hypothesis $h' \subseteq hspace$ that entails the given example with smaller cost.

If such a hypothesis h exists, in line 11 we measure the quality of h by testing how many of the given examples e are correctly predicted by h. This test is performed by repeatedly evaluating an ASP program on bk, h, and the *trace* of the respective example. If the obtained quality is higher than the best previously obtained quality, in lines 15–20 we store the quality and the hypothesis and make a prediction attempt for all test traces. Prediction is performed by evaluating an ASP program on bk, h, and on each trace. If we have correctly predicted the labels of all training examples e, we immediately return the hypothesis after the prediction attempt (because we have no possibility to measure hypothesis improvements beyond this point). In case we do not find a hypothesis that entails all examples, we return the hypothesis that entails most training examples. If we do not find any hypothesis, we return *null*.

For the competition, it was only required that the system makes prediction attempts. To provide a more general approach, our algorithm also returns a hypothesis.

4.1 Soundness and completeness

The INSPIRE system makes a best effort: (i) it learns hypotheses from a single example at a time, (ii) it checks if the found hypothesis covers all examples, (iii) it makes a prediction attempt if the found hypothesis covers more examples than the previously best hypothesis, and (iv) if not all examples were covered, it continues searching for a hypothesis in those examples that were not used for hypothesis search so far.

The hypothesis search and optimization encoding of our system is sound and complete for single examples, that means every hypothesis that is returned will entail the respective example, and the encoding will find all possible minimal hypotheses for a given example. Algorithm 1 ensures that the system will only make a prediction based on a hypothesis that can be verified on more examples than any previously found hypothesis. This approach is a best effort, but it is not sound, as it returns solutions that do not entail all examples. It is also incomplete because the system can find various cheap hypotheses for each example while missing a more expensive hypothesis that entails all examples. *Soundness can be established* by returning only hypotheses that cover all given examples (which would reduce the score in the competition because no partial credit would be gained). Incompleteness is the main disadvantage of our system compared to ILASP, which becomes visible in our analysis of experiments on the instance level, see Sect. 5.2.1. Still, our system provides higher accuracy in the competition settings by solving more instances and by obtaining partial credit for hypotheses that entail only some examples. In a setting where all examples can be represented in the same answer set without interfering with one another, for instance all ILP tasks that can be processed by XHAIL, *completeness can be established* by setting in line 8

$$P_{hs} := bk \cup \{t \mid \langle t, l\rangle \in e\} \cup \{P_{rule}(r) \mid r \in hspace\} \cup \bigcup \{P_{verify}(l) \mid \langle t, l\rangle \in e\}$$

and by modifying Algorithm 1 as follows: remove the example loop (line 4) and the quality check (line 11–14 and line 21). The first hypothesis that is found with this modified approach entails all examples.

5 Evaluation

The INSPIRE system is implemented in Python and uses the version 5.2.2 of the ASP system CLINGO which consists of the grounder GRINGO (Gebser et al. 2011) and the solver

CLASP (Gebser et al. 2012). CLINGO is used for all ASP evaluations shown in Fig. 1. The default configuration of this CLINGO version provides anytime answer sets during optimization due to mixed usage of core-guided and model-guided optimization (Andres et al. 2012; Alviano et al. 2015) in combination with stratification heuristics (Ansótegui et al. 2013). Due to the short timeouts of the ILP competition, we limited ASP computations to $T_{lim} = 5$ seconds each, which was achieved by using the CLINGO parameter `--time-limit=5`. (See Table 4 for experiments with T_{lim}.)

5.1 Comparison system based on ILASP

As there was no other participant in the competition, and as the competition used a novel input format, there was no state-of-the-art system we could use for comparison. Therefore we adapted the state-of-the-art ILASP system (Law et al. 2017, 2016a) version 3.1.0, using a wrapper script. The wrapper converts target and relevant predicates into mode bias commands. Negative examples were specified implicitly in the competition: if a move was not given as valid, it was invalid. Therefore, the wrapper creates for each competition example one positive ILASP example: valid moves are converted to positive atoms, trace atoms are converted into example context [see (Law et al. 2017, Section 5)], and all implicitly given invalid moves are converted into negative atoms.

Example 9 The trace and the valid moves from `#Example(0)` in Fig. 2 are converted into the following ILASP example.

```
#pos(ex0, { valid_move((0,1),0), valid_move((1,0),0), valid_move((1,1),1) },
          { valid_move((0,0),0), valid_move((1,1),0), valid_move((0,0),1),
            valid_move((0,1),1), valid_move((1,0),1) },
          { agent_at((0,0),0).  agent_at((0,1),1). }).
```

To increase ILASP performance, we generate negative example atoms only for those time points where an agent position exists in the trace (most competition examples define 100 time points but use less than 20). We add the directives `#maxv(3)` and `#max_penalty(100)` to configure the ILASP bias; these parameters were found in preliminary experiments: lower values did not yield any hypothesis and higher values yielded much worse performance. In summary, we did our best to make ILASP perform well.

As ILASP realizes two evaluation algorithms that are suitable for different kinds of instances (Mark Law, personal communication), we performed experiments with both algorithms: by ILASP_{2i} we refer to ILASP with parameter `--version=2i`, see (Law et al. 2016a), and by ILASP_3 we refer to ILASP with parameter `--version=3`, see (Law et al. 2017). We provide the ILASP wrapper at https://bitbucket.org/knowlp/inspire-ilp-comp in directory `ilasp-wrapper`.

5.2 Results

We performed experiments on development set (D, 33 instances) and on the test set (T, 45 instances) of the ILP competition, both available on the competition homepage (Law et al. 2016b). Runs were performed for the resource limits of the competition (30 sec, 2 GB) and for higher resource limits (600 sec, 5 GB). Tables show averages over three independent runs for each configuration. The tables contain the number of timeouts (i.e., the system did not terminate within the time limit) both absolute and in percent (lower is better); the accuracy of the last attempt for predicting test traces (this value contains fractions if only some test traces of an instance were predicted correctly); the number of attempts performed, and the average

Table 1 INSPIRE cost parameters for experiments (defaults are in boldface)

System	**cost_negbodyliteral**	**cost_inv**	**cost_inv_pred**	Effect
INSPIRE^{N+}	1	**2**	**2**	Early negation
INSPIRE^{I+}	**2**	1	1	Early predicate invention
INSPIRE	**2**	**2**	**2**	Default parameters
INSPIRE^{N-}	3	**2**	**2**	Late negation
INSPIRE^{I-}	**2**	3	3	Late predicate invention
INSPIRE^{I-N-}	3	3	3	Late predicate invention/negation
INSPIRE^{I--}	**2**	4	4	Even later predicate invention

We indicate the intuitive effect of parameters on the occurrance of rules with negation and predicate invention in the hypothesis search

time T and memory usage M. Experimental results for the INSPIRE system are shown for several cost settings for negation and predicate invention, see Table 1 for an overview.

Table 2 shows results comparing variations of INSPIRE and ILASP$_{2i}$. With respect to *time-outs*, we can see that a timeout of 600 sec yields significantly fewer timeouts and higher accuracy than the competition timeout of 30 sec (both for INSPIRE and ILASP). A timeout occurs if no hypothesis is found and no prediction attempt is made. Therefore, it does not count as a timeout when the INSPIRE system found a hypothesis that predicted some given examples incorrectly and made a (best-effort) prediction attempt on the test data. While ILASP can learn from multiple examples at once, it does not support automatic predicate invention.[3] This explains the lower accuracy and the low number of prediction attempts (ILASP performs a prediction only if all examples are entailed by a hypothesis). The INSPIRE system terminates without timeout for all instances for configurations INSPIRE^{N-} and INSPIRE^{I-N-}. Due to learning from single examples, and due to the upper limit $climit_{max}$ on hypothesis cost, termination does not mean that all examples are covered, therefore termination does not guarantee 100% accuracy (see also Sect. 4.1 about sound- and completeness).

With respect to *accuracy*, the Test dataset appears to be more challenging than the Development dataset. INSPIRE^{N-} was used to participate in the competition, and this configuration achieves the best accuracy for a time limit of 600 sec. For the lower time limit, INSPIRE^{I-N-} (reduced negation and reduced predicate invention) yields the highest accuracy. *Attempts* are generally correlated with accuracy, and it is visible that the limitation in this task is time, not memory.

To find the best configuration, we compare accuracy using a one-tailed paired t-test (paired because we perform experiments on the same instances). INSPIRE^{I-N-} and INSPIRE^{N-} are significantly better than all other configurations with $p < 0.027$, however comparing INSPIRE^{I-N-} and INSPIRE^{N-} shows no significant difference ($p = 0.1$). Increasing predicate invention and negation has the effect of more timeouts and lower accuracy. We analyse results on an instance-by-instance basis in Sect. 5.2.1.

Table 3 shows a comparison of ILASP algorithms. For the test set and high resources, ILASP$_3$ yields slightly higher accuracy than ILASP$_{2i}$, but overall, ILASP$_{2i}$ provides significantly better accuracy than ILASP$_3$ with $p < 0.01$. If we compare Development and Test instances for a time budget of 600 sec, we notice that both ILASP algorithms requires more time but less

[3] Invented predicates can manually be added in an explicit specification of the search space, however we provided ILASP only with a mode bias.

Table 2 Experimental results comparing several INSPIRE configurations (see Table 1) with $T_{lim} = 5$ and ILASP$_{2i}$

DS	Resources	System	Timeout		Accuracy		Attempts	T(s)	M(MB)
			#	%	#	%	#	Avg	Avg
D	30 s, 2 GB	INSPIRE^{N+}	21.0	64	12.0	36	50	20	119
		INSPIRE^{I+}	21.0	64	12.0	36	50	21	113
		INSPIRE	18.0	55	15.0	45	54	17	85
		INSPIRE^{N-}	17.0	52	15.0	45	54	18	87
		INSPIRE^{I-}	17.3	53	15.7	47	52	17	74
		INSPIRE^{I-N-}	**13.0**	**39**	**17.0**	**52**	57	17	74
		INSPIRE^{I--}	15.0	45	**17.0**	**52**	56	17	68
		ILASP$_{2i}$	23.0	70	3.0	9	36	25	44
	600 s, 5 GB	INSPIRE^{N+}	5.0	15	17.4	53	58	195	468
		INSPIRE^{I+}	7.0	21	18.0	55	59	222	400
		INSPIRE	1.0	3	22.6	68	65	107	353
		INSPIRE^{N-}	**0.0**	**0**	**23.4**	**71**	66	64	206
		INSPIRE^{I-}	1.0	3	22.6	68	62	86	266
		INSPIRE^{I-N-}	**0.0**	**0**	**23.4**	**71**	66	47	179
		INSPIRE^{I--}	1.0	3	21.6	65	63	80	249
		ILASP$_{2i}$	4.0	12	14.0	42	47	214	238
T	30 s, 2 GB	INSPIRE^{N+}	27.0	60	15.0	33	63	19	77
		INSPIRE^{I+}	29.0	64	13.0	29	61	20	86
		INSPIRE	27.0	60	15.0	33	63	19	73
		INSPIRE^{N-}	25.0	56	15.0	33	63	19	73
		INSPIRE^{I-}	27.0	60	15.0	33	63	19	68
		INSPIRE^{I-N-}	**21.0**	**47**	**19.0**	**42**	68	17	65
		INSPIRE^{I--}	25.0	56	15.0	33	63	19	66
		ILASP$_{2i}$	39.0	87	3.0	7	48	27	25
	600 s, 5 GB	INSPIRE^{N+}	10.0	22	15.4	34	65	245	640
		INSPIRE^{I+}	7.7	17	18.8	42	73	233	645
		INSPIRE	3.0	7	19.9	44	70	158	471
		INSPIRE^{N-}	**0.0**	**0**	**23.0**	**51**	75	95	311
		INSPIRE^{I-}	1.7	4	16.8	37	67	133	410
		INSPIRE^{I-N-}	**0.0**	**0**	20.9	46	72	65	236
		INSPIRE^{I--}	1.0	2	17.1	38	68	125	382
		ILASP$_{2i}$	13.3	30	13.7	30	59	309	55

The best results in each category are given in bold

memory for the Test instances. This shows that the Test set was structurally different than the Development set, see also Sect. 5.2.1.

Table 4 shows experimental results for several T_{lim} values and the default INSPIRE configuration. The time limit T_{lim} has the primary purpose of providing some, potentially suboptimal, solution within a limited time budget. For participating in the competition, we used $T_{lim} = 5$.

Table 3 Experimental comparison of ILASP algorithms

DS	Resources	System	Timeout		Accuracy		Attempts	T (s)	M (MB)
			#	%	#	%	#	Avg	Avg
D	30 s, 2 GB	ILASP$_{2i}$	**23.0**	**70**	**3.0**	**9**	36	25	44
		ILASP$_3$	23.3	71	**3.0**	**9**	36	26	51
	600 s, 5 GB	ILASP$_{2i}$	**4.0**	**12**	**14.0**	**42**	47	214	238
		ILASP$_3$	7.0	21	11.0	33	44	255	266
T	30 s, 2 GB	ILASP$_{2i}$	**39.0**	**87**	**3.0**	**7**	48	27	25
		ILASP$_3$	40.0	89	2.0	4	47	28	28
	600 s, 5 GB	ILASP$_{2i}$	13.3	30	13.7	30	59	309	55
		ILASP$_3$	**13.0**	**29**	**14.0**	**31**	59	311	88

The best results in each category are given in bold

Table 4 Experimental results with INSPIRE and variations of the time limit parameter T_{lim}

DS	Resources	T_{lim}	Timeout		Accuracy		Attempts	T (s)	M (MB)
			#	%	#	%	#	Avg	Avg
D	30 s, 2 GB	5	18.0	55	15.0	45	54	17	85
		10	18.0	55	15.0	45	54	18	106
		15	18.0	55	15.0	45	54	17	118
		∞	18.0	55	15.0	45	54	18	111
	600 s, 5 GB	5	**1.0**	**3**	**22.6**	**68**	65	107	353
		10	2.0	6	21.8	66	64	120	389
		15	1.3	4	22.3	**68**	65	127	488
		∞	16.0	48	17.0	52	59	320	966
T	30 s, 2 GB	5	27.0	60	15.0	33	63	19	73
		10	27.0	60	15.0	33	63	19	93
		15	27.0	60	15.0	33	63	19	106
		∞	27.0	60	15.0	33	63	19	99
	600 s, 5 GB	5	**3.0**	**7**	**19.9**	**44**	70	158	471
		10	4.0	9	**19.9**	**44**	70	177	579
		15	4.0	9	19.3	43	72	185	733
		∞	26.0	58	16.0	36	64	351	1105

The best results in each category are given in bold

The results in the table show that for the low-resource setting, modifying this parameter has no effect on accuracy. For the high-resource setting, limiting the time yields significantly better accuracy than omitting the timeout ($T_{lim} = \infty$) with $p < 0.01$. Note, that ASP Evaluation is performed multiple times in each run, see Fig. 1 and Algorithm 1, therefore even a low time limit of 5 sec per ASP call can yield an overall timeout of 600 sec.

5.2.1 Instance-by-instance analysis

The competition dataset comprises a development set of 11 instance types, numbered 4 through 12, 14, and 17, moreover there are 15 test instance types, numbered 13, 15, 16, 18, 29, 30, 32 through 35, and 37 through 41. Each instance type exists in three difficulty levels

(easy, medium, and hard), which yields a total of 33 Development and 45 Test instances. In the following, we refer to instance types unless otherwise noted.

In the experiments shown in Table 2, six instances (5, 14, 16, 29, 33, 40) are solved neither by INSPIRE nor by ILASP. A close inspection shows that this is the case for various reasons. One reason is inconsistency in the mode bias: instance 5 contains atoms of form `link`$(\cdot,\cdot)$ in background knowledge but not in the bias, while the bias contains a predicate `full(cell)` that does not exist in the background knowledge. Another reason is, that the time limit T_{lim} is too low: instance 14 can be solved by INSPIRE with $T_{lim} = 60$.

Ten instances (7, 9, 10, 11, 15, 16, 18, 32, 34, 38) yield a (partially) correct prediction with INSPIRE and no prediction with ILASP. From these, four instances (7, 11, 32, 38) require invented predicates in the hypothesis. For two others, ILASP terminates without finding a hypothesis. For the other four instances, ILASP exceeds the timeout. Instance 38 can be solved with default parameters (INSPIRE) but not with reduced predicate invention (INSPIRE^{I--}), while other instances yield results with both configurations. This is an effect of the maximum cost limit setting $climit_{max}$, which makes the approach incomplete if $climit_{max} < \infty$, but yields better performance in the competition setting. The successful hypothesis for Instance 38 comprises six rules where three rules define an invented predicate and one rule requires a negated invented predicate in the body.

Three instances (17, 35, 39) are only solved by ILASP. Of these, instance 39 can be solved with INSPIRE with an increased timelimit $T_{lim} = 60$. The other instances require learning from multiple examples at the same to achieve a correct hypothesis.

For instances 15 and 17, only the hard version can be solved. Hard versions of instances are extensions of easy version with irrelevant mode bias instructions. It seems that in these instances, superfluous predicates made the induction problem easier to solve.

In summary, several factors were important to solve competition instances in a tight time budget, most notably predicate invention, learning from multiple examples, and choosing correct limits for the timeout T_{lim} of intermediate ASP evaluations.

6 Discussion

The INSPIRE system uses a hypothesis search space of stepwise increasing complexity. This is a commonly used approach in ILP for investigating more likely hypotheses first. Usually a very coarse-grained measure of rule complexity is used such as the number of body atoms in a rule. We extend this idea by using an ASP encoding that provides a fine-grained, flexible, easily adaptable, and highly configurable way of describing hypothesis cost for controlling the search space. Our experiments show that the choice of cost parameters has an influence on finding hypotheses that generalize correctly within a given time budget.

We support predicate invention. Increasing the cost of predicate invention in the system (INSPIRE^{I--}) reduces the resource requirements of the system, but unfortunately it also reduces the quality of hypotheses and leads to lower accuracy on the Test dataset. Likewise, increasing predicate invention (INSPIRE^{I+}) leads to lower accuracy because of timeouts that are caused the search space growing too fast. Hence, fine-grained configuration of predicate invention is important for obtaining good results.

Invented predicates with arity one are implicitly created by hypotheses containing only reflexive usage of an invented predicate `inv`, i.e., hypotheses containing only `inv(V,V)` for some variable `V`. Having dedicated unary invented predicates would be a better solution, however this would require to constrain binary invented predicates such that they are used with

two distinct arguments in at least one rule head in the hypothesis (otherwise they are effectively unary predicates). Making a global constraint over the structure of the full hypothesis is currently not possible in INSPIRE, however an approach for such constraints has been described by Athakravi et al. (2015).

Enumerating answer sets of our hypothesis space generation encodings in Algorithm 1 line 6 is computationally cheap compared with hypothesis search (line 9). For finding an optimal hypothesis, the INSPIRE system uses an encoding which requires examples to be represented in disjoint parts of the Herbrand Base, see challenge (C1) in Sect. 1. Therefore, we designed an algorithm that learns from single examples at a time. Our fine-grained hypothesis search space generation (described in Sect. 3) is independent from Algorithm 1 and it is compatible with other approaches for hypothesis search that work with multiple examples and with noisy examples.

The INSPIRE system learns from single examples and sorts them by trace length, which reduces resource consumption in the hypothesis search step. This is a strategy that is specific to the competition: we observed that even short examples often provide sufficient structure for learning the final hypothesis, moreover competition examples were noiseless. We think that future ILP competitions should prevent success of such a strategy by using instances that require learning from all examples at once, e.g., by providing smaller, partial, or noisy examples (even in the non-probabilistic track).

Testing whether a hypothesis correctly predicts an input example is computationally cheap. Therefore, for each newly found hypothesis we perform this check on all examples. If this increases the amount of correctly predicted examples, we make a prediction attempt on the test cases. If all examples were correctly predicted, we terminate the search, because we have no metric for improving the hypothesis after predicting all examples correctly (competition examples are noiseless and our search finds hypotheses with lower cost first).

A major trade-off in our approach is the blind search: we avoid to extract hypotheses from examples as done in the systems XHAIL (Ray 2009) and ILED (Katzouris et al. 2015). This means we rely on the mode bias and on our incrementally increasing cost limit to obtain a reasonably sized search space for hypothesis search.

Note that we did not compare INSPIRE with XHAIL (Ray 2009), MIL (Muggleton et al. 2014), or ILED (Katzouris et al. 2015), partially because these approaches are not compatible with challenge (C1), partially because these approaches are syntactically incompatible with tuple terms (i.e., terms of form `cell((1,2))` which are essential in the background knowledge of the competition.

7 Related work

Inductive Logic Programming (ILP) is a multidisciplinary field and has been greatly impacted by Machine Learning (ML), Artificial Intelligence (AI) and relational databases. Several surveys such as those by Gulwani et al. (2015) and Muggleton et al. (2012) mention about ILP systems and applications of ILP in interdisciplinary areas. Important theoretical foundations of ILP comprise Inverse Resolution and Predicate Invention (Muggleton and Buntine 1992; Muggleton 1995).

Most ILP research has been based on Prolog and aimed at Horn programs that exclude Negation as Failure which provides monotonic commonsense reasoning under incomplete information. Recently, research on ASP-based ILP (Otero 2001; Ray 2009; Law et al. 2014) has made ILP more declarative (no necessity for cuts, unrestricted negation) but also

introduced new limitations (scalability, predicate invention). Predicate invention is indeed a distinguishing feature of ILP: Dietterich et al. (2008) writes that 'without predicate invention, learning always will be shallow'. Predicate invention enables learning an explicit representation of a 'latent' logical structure that is neither present in background knowledge nor in the input, which is related to successful machine learning methods such as Latent Dirichlet Allocation (Blei et al. 2003) and hidden layers in neural networks (LeCun et al. 2015). Muggleton et al. (2015) recently introduced a novel predicate invention method and, to the best of our knowledge for the first time, compared implementations of Metagol in ASP and Prolog. Other ASP-based ILP solvers do not support predicate invention, or they support it only with an explicit specification of rules involving invented predicates (Law et al. 2014). In purely Prolog-based ILP, several systems with predicate invention have been built (Craven 2001; Muggleton 1987; Flach 1993), however these systems also do not support the full power of ASP-based ILP with examples in multiple answer sets (Otero 2001; Law et al. 2014). Note that predicate invention in general is still considered an unsolved and very hard problem (Muggleton et al. 2012).

The TAL (Corapi et al. 2010) ILP system is based on translating an ILP task into an Abductive Logic Programming instance (Kakas et al. 1992), where each rule in the search space is represented by a ground atom that holds a list of body atoms and a list of substitutions of mode bias placeholders with variables in the rule. TAL is Prolog-based and does not instantiate all of these atoms at once. The ASPAL (Corapi et al. 2012) ILP system, similar to TAL, represents rules as single atoms, however ASPAL replaces constants by unique variables in order to instantiate the full search space, followed by an ASP solver call that instantiates these special variables with constants and finds the optimal hypothesis. Our approach provides a declarative and configurable method for enumerating they hypothesis space of an ILP problem. We represent a single hypothesis rule not as a single atom of form $rule(\ldots)$ as done by TAL and ASPAL but as a structured representation in a set of atoms in an answer set. This has the advantage of reducing the size of the instantiation and enables a fast enumeration of all hypothesis candidates. The disadvantage of our approach is, that after enumerating hypothesis candidates we still need another approach for finding the best hypothesis, whereas TAL and ASPAL combine the search for hypothesis candidates with the search for the optimal hypothesis in one representation.

The following extensions of ASPAL, which impose restrictions on hypotheses, have important similarities with our approach. ASPAL was extended in order to perform *minimal revision* of a logic program using ILP (Corapi et al. 2011) by giving hypotheses in the bias a cost according to their edit distance to a given logic program. Revising a logic program was applied to solving ILP tasks in another extension (Athakravi et al. 2014) of ASPAL: initially, a hypothesis that partially covers the examples and has limited complexity is learned, followed by revisions, called *change transactions*, of limited complexity which aim to entail more examples by modifying the hypothesis. This extension of ASPAL limits the search space relative to an intermediate hypothesis, which can rule out the empty hypothesis as solution of certain intermediate steps. Opposed to that, our approach never puts a lower limit on the hypothesis complexity: in INSPIRE, the search space always increases with the number of iterations, while change transactions have the potential to generate big hypotheses based on a sequence of comparatively small searches for locally optimal ILP solutions. ASPAL was extended with a *constraint-driven* bias (Athakravi et al. 2015) where the hypothesis search space, usually given only by a set of head and body mode declarations, can be constrained in a more fine-grained manner. A set of domain-specific rules and constraints (e.g., to require that hypothesis rules with a specific predicate in the head must contain another specific predicate in the body) or generic constraints (e.g., to require that the hypothesis is a stratified

program) imposes hard constraints on hypotheses and these constraints can be formulated differently for each predicate in the mode bias. Different from the constraint-driven bias, we define a preference function and merely delay, but do not completely exclude, hypothesis candidates from the search. Our approach is controlled by cost coefficients and does not permit domain-specific formulation of preferences. Both approaches could be combined to obtain (i) domain-specific control over the overall search space as well as (ii) domain-specific control over search space extension (in case no solution is found in the initial search space). This could be achieved by extending the ASPAL constraint-driven bias language with weak constraints.

A recent application of ASP-based ILP was done by Mitra and Baral (2016), who perform Question Answering on natural language texts. Based on statistical NLP methods, they gather knowledge and learn learning with ILP how to answer questions similar to a given training set. They used XHAIL (Ray 2009) to learn non-monotonic hypotheses in a formalization of an agent theory with events.

8 Conclusion

We created the INSPIRE Inductive Logic Programming system which supports predicate invention and generates the hypothesis search space from the given mode bias using an ASP encoding. This encoding provides many parameters for a fine-grained control over the cost of rules in the search space.

It is useful to have a fine-grained control over the order in which the search space is explored, in particular for controlling negation and predicate invention. Invented predicates are more often useful for abstracting from other concepts, and less often useful if they generate answer sets by introducing additional non-determinism. Similarly, abundance of negation will easily introduce (potentially) undesired non-determinism within the hypothesis. For the ILP competition, the INSPIRE system was configured to first explore hypotheses with mainly positive body literals (see parameter **cost_negbodyliteral**) and hypotheses where invented predicates are used in a stratified way (see parameter **cost_inv_headbodyorder**).

The performance of our system is provided by (i) appropriately chosen cost parameters for the shape of rules in the hypothesis search space, (ii) incremental exploration of the search space, (iii) an algorithm that learns from a single example at a time, and (iv) the usage of modern ASP optimization techniques for hypothesis search. On ILP competition instances, INSPIRE clearly outperforms ILASP, which we adapted to the competition instance format.

The INSPIRE system was created specifically for the first ILP competition, but the fine-grained control over the hypothesis search space generation is a generic method that is independent from the learning algorithm and could be integrated into other systems, for example into XHAIL or ILASP.

Hypothesis candidates are generated in a blind search, i.e., independent from examples. This might seem like a bad choice, however it is a viable option in ASP-based ILP because we found in recent research (Kazmi et al. 2017) that existing methods for non-blind search have major issues which make their usage problematic: the XHAIL algorithm (Ray 2009) produces many redundancies in hypothesis generation, leading to a very expensive search (Induction), while the ILED algorithm (Katzouris et al. 2015) is unable to handle even small inconsistencies in input data, leading to mostly empty hypotheses or program aborts.

We conclude that the good performance of the INSPIRE system is based partially on our novel fine-grained hypothesis search space generation, and partially on the usage of pecu-

liarities of competition instances. To advance the field of Inductive Logic Programming for answer set programming, future competitions should use more diverse instances that disallow finding solutions from single examples, while at the same time requiring the hypothesis for separate examples to be entailed in separate answer sets. To avoid blind search, it will be necessary to improve algorithms and systems to make them resistant to noise and scalable in the presence of large amounts of training examples. We believe that theoretical methods developed in Prolog-based ILP, for example (Muggleton et al. 2015), will be important and useful for advancing ASP-based ILP.

As future work, our hypothesis space generation approach could be integrated into an existing ILP system, for example into the open source XHAIL system by modifying the method `getKernel()` in class `xhail.core.entities.Grounding`. Note, that this would not make XHAIL compatible with examples of the ILP competition due to challenge (C1). Another future work would be to make the INSPIRE input format more generic and to replace the hypothesis optimization encoding with an approach that can process multiple examples at once and noisy examples, e.g., with the full XHAIL encoding or with the encoding from ILASP version 1 (Law et al. 2014).

The INSPIRE system and the ILASP wrapper are open source software and publicly available at https://bitbucket.org/knowlp/inspire-ilp-comp.

Acknowledgements This work has been supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under Grant Agreement 114E777, by the Austrian Science Fund (FWF) under Grant Agreement P27730, and by the Austrian Research Promotion Agency (FFG) under Grant Agreement 861263.

References

Alviano, M., Dodaro, C., Marques-Silva, J., & Ricca, F. (2015). Optimum stable model search: Algorithms and implementation. *Journal of Logic and Computation*, article number exv061.

Andres, B., Kaufmann, B., Matheis, O., & Schaub, T. (2012). Unsatisfiability-based optimization in clasp. In *International conference on logic programming (ICLP), technical communications* (pp. 212–221).

Ansótegui, C., Bonet, M. L., & Levy, J. (2013). SAT-based MaxSAT algorithms. *Artificial Intelligence*, *196*, 77–105.

Apt, K. R., Blair, H. A., & Walker, A. (1988). Towards a theory of declarative knowledge. In J. Minker (Ed.), *Foundations of deductive databases and logic programming* (pp. 89–148). Morgan Kaufmann.

Athakravi, D., Alrajeh, D., Broda, K., Russo, A., & Satoh, K. (2015). Inductive learning using constraint-driven bias. In J. Davis & J. Ramon (Eds.), *Inductive logic programming* (pp. 16–32). Cham: Springer.

Athakravi, D., Corapi, D., Broda, K., & Russo, A. (2014). Learning through hypothesis refinement using answer set programming. In G. Zaverucha, V. Santos Costa, & A. Paes (Eds.), *Inductive logic programming* (pp. 31–46). Berlin: Springer.

Baral, C. (2004). *Knowledge representation, reasoning, and declarative problem solving*. Cambridge: Cambridge University Press.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet allocation. *Journal of Machine Learning*, *3*, 993–1022.

Brewka, G., Eiter, T., & Truszczynski, M. (2011). Answer set programming at a glance. *Communications of the ACM*, *54*(12), 92–103.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., et al. (2012). *ASP-Core-2 input language format*. Tech. rep., ASP Standardization Working Group.

Clocksin, W. F., & Mellish, C. S. (2003). *Programming in PROLOG*. Berlin: Springer.

Corapi, D., Russo, A., De Vos, M., Padget, J., & Satoh, K. (2011). Normative design using inductive learning. *Theory and Practice of Logic Programming*, *11*(4–5), 783–799.

Corapi, D., Russo, A., & Lupu, E. (2010). Inductive logic programming as abductive search. In: International conference on logic programming (ICLP), technical communications, (pp. 54–63).

Corapi, D., Russo, A., & Lupu, E. (2012). Inductive logic programming in answer set programming. In S. H. Muggleton, A. Tamaddoni-Nezhad, & F. A. Lisi (Eds.), *Inductive logic programming* (pp. 91–97). Berlin: Springer.

Craven, M. (2001). Relational learning with statistical predicate invention. *Machine Learning*, *43*, 97–119.
Dietterich, T. G., Domingos, P., Getoor, L., Muggleton, S., & Tadepalli, P. (2008). Structured machine learning: The next ten years. *Machine Learning*, *73*(1), 3–23.
Faber, W., Pfeifer, G., & Leone, N. (2011). Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, *175*(1), 278–298.
Flach, P. A. (1993). Predicate invention in inductive data engineering. In *European conference on machine learning (EMCL)* (pp. 83–94).
Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012a). *Answer set solving in practice*. Morgan Claypool.
Gebser, M., Kaminski, R., König, A., & Schaub, T. (2011). Advances in gringo series 3. In *International conference on logic programming and non-monotonic reasoning (LPNMR)* (pp. 345–351).
Gebser, M., Kaufmann, B., & Schaub, T. (2012b). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, *187–188*, 52–89.
Gelfond, M., & Kahl, Y. (2014). *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge: Cambridge University Press.
Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In *International conference and symposium on logic programming (ICLP/SLP)* (pp. 1070–1080).
Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., & Zorn, B. (2015). Inductive programming meets the real world. *Communications of the ACM*, *58*(11), 90–99.
Kakas, A. C., Kowalski, R. A., & Toni, F. (1992). Abductive logic programming. *Journal of Logic and Computation*, *2*(6), 719–770.
Katzouris, N., Artikis, A., & Paliouras, G. (2015). Incremental learning of event definitions with inductive logic programming. *Machine Learning*, *100*(2–3), 555–585.
Kazmi, M., Schüller, P., & Saygn, Y. (2017). Improving scalability of inductive logic programming via pruning and best-effort optimisation. *Expert Systems with Applications*, *87*, 291–303.
Law, M., Russo, A., & Broda, K. (2014). Inductive learning of answer set programs. In *European conference on logics in artificial intelligence (JELIA)* (pp. 311–325).
Law, M., Russo, A., & Broda, K. (2015). Learning weak constraints in answer set programming. *Theory and Practice of Logic Programming*, *15*(4–5), 511–525.
Law, M., Russo, A., & Broda, K. (2016a). Iterative learning of answer set programs from context dependent examples. *Theory and Practice of Logic Programming*, *16*(5–6), 834–848.
Law, M., Russo, A., & Broda, K. (2017). *Inductive learning of answer set programs v3.1.0 user manual*. Tech. rep., Imperial College of Science, Technology and Medicine, Department of Computing.
Law, M., Russo, A., Cussens, J., & Broda, K. (2016b). *The 2016 competition on inductive logic programming*. Retrieved March 29, 2017, http://ilp16.doc.ic.ac.uk/competition.
LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.
Lifschitz, V. (2008). What is answer set programming? In *AAAI conference on artificial intelligence* (pp. 1594–1597).
Mitra, A., & Baral, C. (2016). Addressing a question answering challenge by combining statistical methods with inductive rule learning and reasoning. In D. Schuurmans & M. P. Wellman (Eds.), *Association for the advancement of artificial intelligence* (pp. 2779–2785). AAAI Press.
Muggleton, S. (1987). Duce, an oracle-based approach to constructive induction. In *International joint conference on artificial intelligence (IJCAI)* (pp. 287–292).
Muggleton, S. (1995). Inverse entailment and Progol. *New generation computing*, *13*(3–4), 245–286.
Muggleton, S., & Buntine, W. (1992). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the fifth international conference on machine learning* (pp. 339–352).
Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, *19*, 629–679.
Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K., et al. (2012). ILP turns 20: Biography and future challenges. *Machine Learning*, *86*(1), 3–23.
Muggleton, S. H., Lin, D., Pahlavi, N., & Tamaddoni-Nezhad, A. (2014). Meta-interpretive learning: Application to grammatical inference. *Machine Learning*, *94*(1), 25–49.
Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2015). Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, *100*(1), 49–73.
Otero, R. P. (2001). Induction of stable models. In *Conference on inductive logic programming* (pp. 193–205).
Ray, O. (2009). Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, *7*, 329–340.
Sakama, C., & Inoue, K. (2009). Brave induction: A logical framework for learning from incomplete information. *Machine Learning*, *76*, 3–35.