Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning
Special Session on KR and Machine Learning

Learning First-Order Representations for Planning from Black-Box States: New Results

Ivan D. Rodriguez[1] , **Blai Bonet**[1] , **Javier Romero**[2] , **Hector Geffner**[1,3]
[1]Universitat Pompeu Fabra, Spain
[2]University of Potsdam, Germany
[3]Institució Catalana de Recerca i Estudis Avançats (ICREA)
{ivandanielra,bonetblai}@gmail.com, javier@cs.uni-potsdam.de, hector.geffner@upf.edu

Abstract

Recently Bonet and Geffner have shown that first-order representations for planning domains can be learned from the structure of the state space without any prior knowledge about the action schemas or domain predicates. For this, the learning problem is formulated as the search for a simplest first-order domain description D that along with information about instances I_i (number of objects and initial state) determine state space graphs $G(P_i)$ that match the observed state graphs G_i where $P_i = \langle D, I_i \rangle$. The search is cast and solved approximately by means of a SAT solver that is called over a large family of propositional theories that differ just in the parameters encoding the possible number of action schemas and domain predicates, their arities, and the number of objects. In this work, we push the limits of these learners by moving to an answer set programming (ASP) encoding using the CLINGO system. The new encodings are more transparent and concise, extending the range of possible models while facilitating their exploration. We show that the domains introduced by Bonet and Geffner can be solved more efficiently in the new approach, often optimally, and furthermore, that the approach can be easily extended to handle partial information about the state graphs as well as noise that prevents some states from being distinguished.

1 Introduction

One of the main research challenges in AI is how to bring together learning and reasoning, and in particular, learning approaches that can deal with non-symbolic inputs and reasoning approaches that require first-order symbolic inputs (Lake et al. 2017; Marcus 2018; Pearl 2018; Darwiche 2018). On one hand, pure data-based approaches like deep learning produce black boxes that are hard to understand and which do not generalize well; on the other, model-based approaches, in particular those that require first-order representations, require models which are normally crafted by hand.

A concrete challenge for making the best of both data-based learners and model-based reasoners is to learn from data the type of representations that are required by solvers (Geffner 2018). In this work we are particularly interested in learning the first-order symbolic representations that are used in classical planning in languages such as PDDL (McDermott 2000; Haslum et al. 2019). These languages have evolved throughout years of research and exhibit a number of benefits concerning both generalization and reusability. Planning problems in PDDL-like languages are expressed in two parts: a domain D that expresses the action schemas and their preconditions and effects in terms of a fixed set of domain predicates, and instance information I that details the objects (names) and the ground atoms that are true in the initial situation and the goal. The domain D and the instance information I, together, define a complete planning instance $P = \langle D, I \rangle$. There is, however, an infinite collection of planning instances that can be defined over the same domain, and the action schemas and predicates provide a language for capturing what is common in all of them. Thus, if one manages to learn the domain predicates and schemas from some instances, one learns a representation that applies to all other domain instances as well.

Two approaches have been recently proposed for learning first-order planning representations from non-symbolic data. In one case, the input data corresponds to one or more state graphs G_i assumed to originate from hidden planning instances $P_i = \langle D, I_i \rangle$ that need to be uncovered. In these input graphs, the nodes correspond to different states, and the edges correspond to the possible state transitions. The states are black boxes and nothing is assumed to be known about their structure except that states associated with different nodes must be different (Bonet and Geffner 2020). In the second case, the input involves state trajectories associated with an instance with each state represented by an image. Propositional and first-order action representations are then obtained through the use of a class of (variational) autoencoders (Kingma and Welling 2014), where the input images must be recovered in the output of a deep neural network by going through a categorical representation (Asai 2019). The two approaches appeal to different principles for uncovering the representations: in the first case, the learned representations must recover the input graphs; in the second, the images associated with the states. The approach based on *graph recovery* yields crisp symbolic representations that match the intended models well; the second approach based on *image recovery* yields symbolic representations that are less crisp but which are grounded on the images, and make less assumptions about the inputs than the graph recovery approach that requires the set of trajectories (graphs) to be complete and noise-free.

In this work, we explore variations and extensions of the approach proposed by Bonet and Geffner that make it more

scalable and more robust. In their work, the search for the simplest PDDL models $P_i = \langle D, I_i \rangle$ that account for the input graphs G_i is formulated and solved, approximately, by means of a SAT solver that is called over a large family of propositional theories differing in the parameters encoding the possible number of action schemas and domain predicates, their arities, and the number of objects. We move from a low-level SAT encoding to a high-level *answer set programming (ASP)* encoding (Brewka, Eiter, and Truszczyński 2011; Lifschitz 2019) using the CLINGO system (Gebser et al. 2012; Gebser et al. 2019). The new encoding is more transparent and more concise, and extends the range of possible models while facilitating their exploration. We show that the domains introduced by Bonet and Geffner can be solved *more efficiently* in the new approach, in many cases optimally, and furthermore that simple extensions suffice to overcome some of the limitations, like the assumption that the input graphs are complete and noise-free. Indeed, the new encodings can handle naturally *partial information about the state graphs*, as well as *noise that prevents some states from being distinguished from other states.*

The paper is organized as follows. We first review related work and the learning formulation advanced by Bonet and Geffner. Then we solve this formulation via ASP, consider a number of extensions and optimizations, and present the experimental results and the extensions for dealing with incomplete and noisy samples of the input graphs.

2 Related Work

The paper builds on prior work by Bonet and Geffner (2020) and is related to the work by Asai (2019), both focused on the problem of learning first-order symbolic representations of planning domains from non-symbolic data. The language of these representations is a subset of PDDL which is suitable for transferring knowledge learned from some planning instances to others. Another form of knowledge that can be transferred among instances of the same domain is given by *general policies or plans* (Srivastava, Immerman, and Zilberstein 2008; Bonet, Palacios, and Geffner 2009; Hu and De Giacomo 2011; Belle and Levesque 2016). A general policy provides a full-detailed strategy for solving a collection of problems. Approaches for learning such general plans from some instances have been developed as well (Khardon 1999; Martín and Geffner 2004; Fern, Yoon, and Givan 2003; Bonet, Francès, and Geffner 2019; Francès, Bonet, and Geffner 2021), some of them relying on deep learning techniques (Toyer et al. 2018; Bueno et al. 2019; Issakkimuthu, Fern, and Tadepalli 2018; Bajpai, Garg, and Mausam 2018). Since these approaches require a first-order representation of the planning domains, learning these representations provides a necessary step for learning the general policies.

Deep reinforcement learning (DRL) methods (Mnih et al. 2015) have been used to learn general policies over high-dimensional perceptual spaces without using or producing symbolic knowledge (Groshev et al. 2018; Chevalier-Boisvert et al. 2019; François-Lavet et al. 2019). Yet by not constructing first-order representations involving objects and relations, their ability to generalize appears to be limited. Recent work in deep symbolic relational reinforcement learning (Garnelo and Shanahan 2019; Shanahan et al. 2020) attempts to account for objects and relations through the use of suitable neural architectures, but the gap between the low-level techniques used and the high-level representations required is large. More recently, model-based DRL approaches have been shown to learn informative latent representations and have achieved considerable success in specific settings and video-games, but their abilities for generalization in the presence of new objects has not been explored (Hafner et al. 2019; Schrittwieser et al. 2020; Hafner et al. 2020).

Finally, there is a large body of work on learning first-order planning representations given partial knowledge about instances and domains (Diuk, Cohen, and Littman 2008; Yang, Wu, and Jiang 2007; Arora et al. 2018; Aineto et al. 2019; Cresswell, McCluskey, and West 2013); e.g., learning the domain's action schemas given the predicates, sampled state trajectories, and the structure of the states. These works however do not address the learning of the predicates.

3 Formulation of the Learning Problem

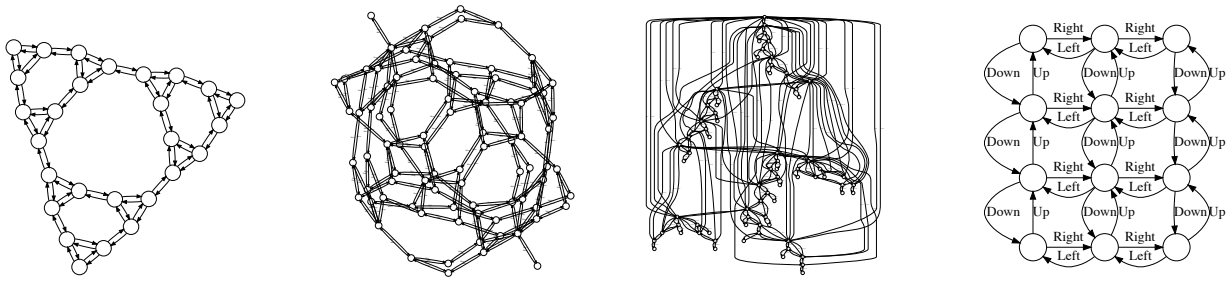
We follow the mathematical formulation proposed by Bonet and Geffner, and introduce then extensions and variations:

- Given a labeled graph $G = \langle V, E, L \rangle$, where the nodes n correspond to the different (black box) states, and the edges (n, n') in E with label $l \in L$, correspond to state transitions produced by an action with label l,
- find a "simplest" planning instance $P = \langle D, I \rangle$ such that the state graph $G(P)$ defined by P and G are isomorphic.

Multiple input graphs $G_1, \ldots, G_k$ are handled in a similar way by finding the simplest common domain D and instances $P_i = \langle D, I_i \rangle$ whose state graphs $G(P_i)$ match the given graphs G_i. The formulation assumes that the input graphs G_i are *complete* in the sense that they do not miss any edge, that they are *noise-free* in the sense that no edges in them are wrong, and that different nodes stand for different states (different sets of true literals). The input graphs G_i are actually *labeled graphs* with *action labels* used to add information about the actions in the input. These action labels do not convey information about the structure of the action schemas, nor about their arities or the predicates involved in their preconditions and effects. Figure 1, from Bonet and Geffner, shows the input graphs considered in their paper corresponding to instances of the Towers of Hanoi, Gripper, Blocksworld, and Grid. The graphs displayed are the sole inputs to their representation learning scheme and ours.

3.1 Formalization: Learning and Verification

A (classical) planning instance is a pair $P = \langle D, I \rangle$ where D is a first-order planning domain and I represents the instance information. The planning domain D contains a set of predicate symbols and a set of action schemas with preconditions and effects given by atoms $p(x_1, \ldots, x_k)$ or their negations, where p is a domain predicate and each x_i is a variable representing one of the arguments of the action schema. The



(a) Towers of Hanoi (1 label) (b) Gripper (3 labels) (c) Blocksworld (3 labels) (d) 4×3 Grid (4 labels)

Figure 1: Input data for learning representations in 4 planning domains from Bonet and Geffner (2020). Graphs contain the edge labels.

instance information is a tuple $I = \langle O, Init, Goal \rangle$ where O is a (finite) set of object names c_i, and $Init$ and $Goal$ are sets of ground atoms $p(c_1, \ldots, c_k)$ or their negations. The actual name of the constants in O is irrelevant and can be replaced by numbers in the interval $[1, N]$ where $N = |O|$. Similarly, goals are included in I to keep the notation consistent with planning practice, but they play no role in the formulation.

A planning problem $P = \langle D, I \rangle$ defines a *labeled* graph $G(P) = \langle V, E, L \rangle$ where the nodes n in V correspond to the states $s(n)$ over P, and there is an edge (n, n') in E with label a, (n, a, n'), if the state transitions $(s(n), s(n'))$ is enabled by a ground instance of the schema a in P. The states $s(n)$ are maximally consistent sets of ground literals over P that comply with: $Init$ corresponds to state $s(n_0)$ for some node n_0, states $s(n)$ and $s(n')$ are different if $n \neq n'$, and $(s(n), s(n'))$ is enabled by a ground action iff the preconditions of the action are true in $s(n)$, the effects are true in $s(n')$, and literals whose complements are not made true by the action have the same value in $s(n)$ and $s(n')$.

Definition 1 (Bonet and Geffner, 2020)**.** *The **learning problem** is to find the simplest instances $P_i = \langle D, I_i \rangle$ that account for a set of **input labeled graphs** G_i, $i = 1, \ldots, k$.*

Here an instance P **accounts for** a labeled graph G when there is a 1-1 and onto function h between the reachable nodes in $G(P)$ and those in G, and a 1-1 and onto function g between the action labels in $G(P)$ and those in G, such that (n, a, n') is a labeled edge in $G(P)$ iff $(h(n), g(a), h(n'))$ is a labeled edge in G. In words, P accounts for a graph G when the graph $G(P)$ and G match in this way.

This is a learning problem and not a synthesis problem; namely, the representations $P_i = \langle D, I_i \rangle$ are not deducible from the input graphs; they are inferred under suitable regularity (simplicity) assumptions, and the learned domains D are expected to **generalize** to other instances.

Given a suitable definition of "simplest", the learning problem becomes a combinatorial optimization problem, as the space of possible domain representations D is made finite once a bound on the number of action schemas, predicates, arguments (arities), and objects is defined. Testing the **generalization** of the learned domain in other instances is another combinatorial problem:

Definition 2. *The **verification problem** is to test whether there are instances $P'_i = \langle D, I'_i \rangle$ over learned domain D that account for a set of **testing labeled graphs** G'_i, $i = 1, \ldots, k'$.*

The verification problem is a subproblem of the learning problem where the domain D is not learned but fixed, and just the instance information needs to be inferred.

4 Basic ASP Encoding

Bonet and Geffner (2020) address the learning and verification problems above as **SAT problems** using a vector of hyperparameters α to represents the *exact* number of action schemas and the arity of each one of them, the number of predicate symbols and the arity of each one of them, and so on. Since the true value of these parameters is not known, they generate a propositional theory $T_\alpha(G)$ for each possible vector of hyperparameters α within certain bounds, where G is the input graph (a single input graph is shown to be enough for learning the representations). This number of propositional theories can be very high.

Our move from SAT to ASP is aimed at getting more compact and transparent encodings that are easier to understand and explore, and that for this reason, can also deliver superior performance. We present the resulting encodings in two parts, first a basic ASP encoding that is easier to follow and understand, and then the required optimizations. In these encodings, the learning problems considered by Bonet and Geffner are solved by a handful of calls to the ASP solver CLINGO (Gebser et al. 2012; Gebser et al. 2019), as opposed to the thousands of calls required in the SAT encodings. Moreover, once a preference ordering on solutions is defined, some of these problems are shown to be solved *optimally*, meaning that there is no simpler representation compatible with the input data.

4.1 Program

The ASP code ASP(G) for learning a first-order instance $P = \langle D, I \rangle$ from a single input graph G is shown in Figure 2. The code can be easily generalized to learn the instances $P_i = \langle D, I_i \rangle$ from multiple input graphs G_i. The graph G is assumed to be encoded using the atoms `node(S)` and `tlabel(T,L)` where `S` and `T=(S1,S2)` denote nodes and transitions in the graph G, and `L` denotes the corresponding action label. The resulting lifted action schemas are encoded via the `prec/3` and `eff/3` atoms (the integer at the end denotes the arity of the predicate), while the factored states of the nodes `S`, via the `val/2` and `val/3` atoms, the first for static atoms and the second for dynamic ones. The main

constraints are at the bottom of the program in Fig. 2: transitions `(S1,S2)` are assigned exactly to one ground action through the atoms `next(A,OO,S1,S2)` where `OO` are the objects that instantiate the arguments of the action schema `A`. As a result, the value of the ground atoms in the states `S1` and `S2` has to be compatible with this ground action. In addition, the ground action `A(OO)` has to be applicable in `S1`, and if it is applicable in a node `S1'`, then there has to be a node `S2'` such that `next(A,OO,S1',S2')` is true. Also, the sets of literals true in a node (the states associated to nodes) have to be different for different nodes. The static predicates and atoms refer to those that control the grounding; i.e., they appear in the precondition of action schemas but do not appear in the effects, so their value do not depend on the state.

The first part of the program (lines 1–11) sets up the bounds on the domain parameters: max number of predicates (5) and static predicates (2), and max numbers of effects and preconditions per action schema (6). In addition, the arity of actions is bounded by 3, and the arity of predicates by 2. The number of action schemas is set to the number of action labels, and the number of objects is fixed to the constant `num_objects` that is passed to the solver. In our experiments, the solver is instantiated with values $1, \ldots, 10$ for this parameter; i.e., for each input graph, the solver is called up to 10 times.

The second part (lines 13–31) sets up the action schemas and their lifted preconditions and effects, while the third part (lines 33–48) encodes the grounding: for a given number of objects, it generates the possible object tuples `OO` that can instantiate the arguments of action schemas and predicates. Pairs `(A,OO)` and `(P,OO)` denote ground actions and ground atoms. Atoms `map(VV,OO1,OO2)` tell that the variable list `VV` instantiates to `OO2` for the ground instantiation `OO1` of the action arguments. Truth values `V` of ground atoms `(P,OO)` in the state `S` are encoded with atoms `val((P,OO),S,V)`, with the state `S` omitted for static predicates.

The last part (lines 50–59) encodes the main constraints: 1) if the ground action `(A,OO)` is assigned to edge `(S1,S2)`, it must be applicable in `S1`, the values of the ground literals in `S1` and `S2` must be consistent with the action effects, and the action labels must coincide, 2) if the ground action `(A,OO)` is applicable in a node, it must be associated with a unique outgoing edge, and 3) the sets of true ground literals for different nodes must be different. This fragment of the program ASP(G) makes an *assumption* which is most often but not always true; namely, that if two ground instances of an action schema are applicable in a state, their application leads to different states.[1] Provided with this assumption, the correctness of the encoding can be expressed as:

Theorem 3. *Let G be an input graph encoded with the `node/2` and `tlabel/2` atoms. Then M is an answer set of the program* ASP(G) *iff there is a planning instance $P = \langle D, I \rangle$ that accounts for G and is compatible with the given bounds; D and I can be read off from M.*

[1] An schema with arguments x and y, effects $p(x)$ and $p(y)$, and no preconditions, for example, would violate this assumption. Thanks to Andres Occhipinti for pointing this out.

5 Extensions and Optimization

The code shown in Fig. 2 captures a correct encoding of the learning problem but it is not optimized for performance. We discuss next some variations and extensions. The code for these extensions is shown in Fig. 3.

Optimization. The program ASP(G) for a given graph G usually has many models. Following Occam's razor, the models that are simpler are more likely to result in action schemas that generalize to larger instances and hence to pass the verification test. Models are thus ranked using four numerical criteria ordered **lexicographically** from most important to less important as follows: A) sum N_a of the action schemas arities, B) sum N_p of the arities of non-static predicates, C) sum N_s of the arities of static predicates, D) maximum number N_g of true ground atoms per state.

If $V(M) = (N_a, N_p, N_s, N_g)$ is the resulting numerical cost vector for model M, and $V(M') = (N'_a, N'_p, N'_s, N'_g)$ is the vector for model M', M is preferred to M' if $V(M)$ is lexicographically smaller than $V(M')$. A model M is **optimal** or **simplest** if there is no model M' preferred to M. This exact optimization criterion is given to CLINGO along with the code (lines 1–5 in Fig. 3). For a given **time window**, CLINGO may terminate earlier with an optimal solution, or run out of time, returning the **best solution** found that far.

Invariants. One way to speed up SAT and ASP solvers is by adding implicit constraints. State invariants express formulas that are true in all (reachable) states, and they are implied by the truth valuation of the initial state and the structure of the action schemas. One particular type of invariants is given by sets of atoms R such that *exactly-1* of the atoms in R is true in every state. An invariant of this form, for example in Blocksworld without an arm, is given by the sets of atoms $\{on(x, y) : y\} \cup \{ontable(x)\}$ for any block x, where y ranges over all blocks. The variable x is the **free variable** of the **invariant schema**, while other variables like y range over their possible instantiations.

We extend the basic ASP encoding so that *exactly-1* invariant schemas can be constructed and enforced automatically during the search. This is achieved by introducing atoms `schema(N,P,I)` where `N` is an index over a max number of lifted invariants, `P` is a predicate, and `I` is the index of the argument of P that represents the free variable if P is binary (for unary predicates there is no choice). The solver is free to choose the atoms `schema(N,P,I)` that are true, and the resulting constraints are enforced as invariants; i.e., they must be true in all states. Furthermore, to force some invariants to be true, every *dynamic binary predicate* is constrained to be part of some invariant. This is a common situation where dynamic binary predicates like $p(x, y)$ are used to encode multivalued state variables. Lines 12–31 in Fig. 3 implement the construction and enforcement of invariants.

Other extensions. The actual code of the program ASP(G) incorporates other extensions like constraints for symmetry breaking, special rules for handling cycles of size two in the input graph, and transformations for treating all action schemas (resp. predicates) as if they are all of the same

```
% Constants
#const max_predicates=5.
#const max_static_predicates=2.
#const max_effects=6.           % for each action
#const max_precs=6.             % for each action

% Actions, predicates, static predicates and objects
action(L) :- tlabel(T,L).  { a_arity(A,1..3) } = 1 :- action(A).
 pred(1..max_predicates).  { p_arity(P,1..2) } = 1 :-   pred(P).
p_static(max_predicates-max_static_predicates+1..max_predicates).
object(1..num_objects).

% Tuples of variables for lifted effects and preconditions
argtuple((V1,  ),1) :- V1=1..3. argtuple((V1,V2),2) :- V1=1..3, V2=1..3.

% Generate lifted preconditions and effects (at least 1) of action schemas
  { prec(A,(P,T),0..1) : p_arity(P,AR), argtuple(T,AR)                }   max_precs :- action(A).
1 { eff(A,(P,T),0..1) : p_arity(P,AR), argtuple(T,AR), not p_static(P) } max_effects :- action(A).

% Check that variables mentioned in precs and effects are action arguments
:-  eff(A,(_,(V, )),_), a_arity(A,ARITY), ARITY < V.
:-  eff(A,(_,(V,_)),_), a_arity(A,ARITY), ARITY < V.
:-  eff(A,(_,(_,V)),_), a_arity(A,ARITY), ARITY < V.
:- prec(A,(_,(V, )),_), a_arity(A,ARITY), ARITY < V.
:- prec(A,(_,(V,_)),_), a_arity(A,ARITY), ARITY < V.
:- prec(A,(_,(_,V)),_), a_arity(A,ARITY), ARITY < V.

% Tuples of objects for grounding the action schemas and atoms
objtuple((O1,     ),1) :- object(O1).
objtuple((O1,O2   ),2) :- object(O1), object(O2).
objtuple((O1,O2,O3),3) :- object(O1), object(O2), object(O3).

% Possible values of ground atoms in the states associated to nodes
{ val((P,OO),  0..1) } = 1 :- p_arity(P,ARITY),     p_static(P), objtuple(OO,ARITY).
{ val((P,OO),S,0..1) } = 1 :- p_arity(P,ARITY), not p_static(P), objtuple(OO,ARITY), node(S).

% Map selects grounding of lifted atoms in schema from grounding of action arguments
map( (1,),     (O1,),  (O1,)) :-      objtuple((O1,),1).
...
map((3,3),(O1,O2,O3),(O3,O3)) :- objtuple((O1,O2,O3),3).

% Check preconditions: ground action A(OO1) is applicable in node S
appl(A,OO1,S) :- action(A), a_arity(A,ARITY), objtuple(OO1,ARITY), node(S),
                 val((P,OO2),  V) : prec(A,(P,T),V), map(T,OO1,OO2),     p_static(P);
                 val((P,OO2),S,V) : prec(A,(P,T),V), map(T,OO1,OO2), not p_static(P).

% If ground action A(OO) applicable in S1, assigned to some edge (S1,S2) with same label
{ next(A,OO,S1,S2) : tlabel((S1,S2),A) } = 1 :- appl(A,OO,S1).

% Every edge is assigned to a ground action with the same label
:- tlabel((S1,S2),A), { next(A,OO,S1,S2) } ≠ 1.

% Effects and inertia
:- eff(A,(P,T),V), next(A,OO1,S1,S2), map(T,OO1,OO2), val((P,OO2),S2,1-V).
:- tlabel((S1,S2),_), val(K,S1,V), val(K,S2,1-V), not caused(S1,S2,K).
caused(S1,S2,(P,OO2)) :- eff(A,(P,T),V), next(A,OO1,S1,S2), map(T,OO1,OO2).

% Different nodes are different states
:- node(S1), node(S2), S1 < S2, val((P,T),S2,V) : val((P,T),S1,V).
```

Figure 2: Base code of ASP program ASP(G) for learning a first-order planning instance $P = \langle D, I \rangle$ from a single input graph G. The graph G is assumed to be encoded with atoms `node(S)` and `tlabel(T,L)` where `T=(S1,S2)` stands for the transitions in the graph and `L` for the corresponding action label. See text for explanations.

Domain	#labels	#obj	#nodes	#edges
blocks1-2 (arm, 2 blocks)	4	2	5	8
blocks1-3 (arm, 3 blocks)	4	3	22	42
blocks1-4 (arm, 4 blocks)	4	4	125	272
blocks1-5 (arm, 5 blocks)	4	5	866	2,090
blocks2-2 (2 blocks)	3	2	3	4
blocks2-3 (3 blocks)	3	3	13	30
blocks2-4 (4 blocks)	3	4	73	240
blocks2-5 (5 blocks)	3	5	501	2,140
grid-v0-3x4	4	4	12	34
grid-v0-4x4	4	4	16	48
grid-v0-5x6	4	6	30	98
grid-v1-3x4	2	4	12	34
grid-v1-4x4	2	4	16	48
grid-v1-5x6	2	6	30	98
gripper-2 (2rooms + 2balls)	3	4	28	76
gripper-3 (2rooms + 3balls)	3	5	88	280
gripper-4 (2rooms + 4balls)	3	6	256	896
hanoi-3x3 (3disks + 3pegs)	1	6	27	78
hanoi-3x4 (3disks + 4pegs)	1	7	81	240
hanoi-4x3 (4disks + 3pegs)	1	7	74	336

Table 1: Data for the graphs $G(P)$ of the instances P used for learning the domains and for their verification: numbers of action labels, nodes, and edges. Number of objects in P shown as well.

(max) arity (to reduce grounding size). [2]

6 Experimental Results

We test the performance of the program ASP(G) with all the extensions above. The program accepts a single graph G and outputs a model (solution) from which a first-order planning instance $P = \langle D, I \rangle$ that matches G can be read off.

We consider five domains: two versions of Blocksworld (with and without an arm), Towers of Hanoi, Gripper and Grid; all from Bonet and Geffner (2020), except Blocksworld with an arm. The experiments were performed on Amazon EC2's `c5n.12xlarge` nodes with a limit of 16Gb of memory, 2 hours for learning, and 1 hour for verification. Recall that verification is a combinatorial problem similar to the learning problem but with the learned model known and fixed (Section 3.1). The data about the instances used is shown in Table 1. For each instance P in the table, the graph $G(P)$ is used to learn a first-order model, and the learned model (if any) is then verified on the other instances P' of the same domain that are listed in the table. The solver CLINGO was run with 8 threads.

Table 2 shows the results of the ASP-based learner on these instances. As mentioned above, the solver is called multiple times with different number of objects, from 1 up to 10, and the models with the smallest number of objects that passed the verification are reported for each domain (except for Grid where models with a smaller number of objects than expected also generalized, and for Blocks where in some instances no solution that generalized was found). Bounds

[2]The code and the relevant data are available at https://github.com/bonetblai/learner-strips/tree/master/asp.

			Time to find model			
Domain	#obj	#mod	First	Best	Ver	Opt
blocks1-3	3	8	35.08	93.85	✗	✓
blocks1-4	4	0	—	—	—	—
blocks2-3	3	15	3.86	10.84	✗	✓
blocks2-4	4	26	550.1	2,564.2	✓	✓
grid-v0-3x4	2	13	2.9	13.9	✓	✓
grid-v0-3x4	3	14	5.5	23.1	✓	✓
grid-v0-3x4	4	20	3.2	24.1	✓	✓
grid-v1-3x4	3	13	1.0	105.2	✓	✓
grid-v1-3x4	4	12	1.2	12.4	✓	✓
gripper-2	4	37	356.3	7,200.0	✓	✗
gripper-3	4	1	6,540.2	7,200.0	✓	✗
hanoi-3x3	6	6	664.9	7,200.1	✓	✗

Table 2: Results of ASP(G) on the instances in Table 1. Columns show the instance used for learning, the number of objects found, and times for the first and best solutions found, and whether the best solution verified and was proved to be optimal (simplest). The column "#mod" shows the number of models found in the way to the best model. Times are in seconds.

on the number of action schemas, predicates and their arities are those expressed in the code shown in Figure 2. The columns "Ver" and "Opt" in the table refer to whether the model found verified (generalized to the other instances in Table 1), and whether it was proved to be optimal (i.e., simplest according to the optimization criterion). The table also reports the time to find a model for each instance (time to first model), and the time to find the best model within the time window (2h). A time of 7,200 seconds indicates that the best model was not proved to be optimal in the time window. The table also displays the number of models found by the solver in the optimization process, each model being better than the previous one.

The key observation that can be drawn from Table 2 is that models that generalize to the other instances of the same domain are found in all of the domains, except in Blocks-1 (Blocks with an arm). Moreover, some models that are shown to be optimal do not generalize (Blocks-1-3, Blocks-2-3), while models that are not proved to be optimal, do (Gripper, Hanoi). The reason for the former is that the instances used for learning are too small, which leads to action schemas that do not generalize to larger instances. The reason for the latter is that strict optimality is not a condition for generalization. In the case of the Blocks-1 domain, larger instances like Blocks-1-4 timed out. Regarding the size of the ground programs, they feature between 30K to 70K rules and constraints in Grid and Gripper-2, and between 1M to 2M in Blocks, Gripper-3 and Hanoi. Similarly, the number of variables ranges between 50K to 165K in the former, and between 300K to 800K in the latter.

The improvements in relation to the results reported by Bonet and Geffner (2020) using a SAT approach are significant. As it was mentioned before, for each input G, they need to call the SAT solver over the theory $T_\alpha(G)$ for each possible vector of hyperparameters α, which means a number of calls that range from 6,390 (Hanoi, 1 label) to 19,050

```
% Optimization
#minimize { 1+N@4, A : a_arity(A, N)                  }.
#minimize { 1+N@3, P : p_arity(P, N), not p_static(P) }.
#minimize {   N@2, P : p_arity(P, N),     p_static(P) }.
#minimize {   N@1, N : state_bound(N)                 }.

% Bound number of true atoms per state
#const max_true_atoms_per_state=10.
{ state_bound(1..max_true_atoms_per_state) } = 1.
{ val((P,OO),S,1) : pred(P), not p_static(P), objtuple(OO,2) } N :- node(S), state_bound(N).

% Choose number and type of invariants
#const num_invariants=1.
inv(1..num_invariants).

% Schemas for invariants
{ schema(N,P,1) } :- inv(N), pred(P), p_arity(P,1).
1 { schema(N,P,2..3) : pred(P), p_arity(P,2), not p_static(P) } :- inv(N), bin_preds.
inv_non_empty(N) :- inv(N), pred(P), schema(N,P,1..3).
bin_preds :- pred(P), p_arity(P,2), not p_static(P).

% Each non-static binary predicate must appear in some invariant
inv_used_pred(P) :- pred(P), p_arity(P,2), not p_static(P), inv(N), schema(N,P,2..3).
:- pred(P), p_arity(P,2), not p_static(P), not inv_used_pred(P).

% Enforce invariants at states
{  val((P, (O,O)),  1) :             schema(N,P,1),     p_static(P); % { P(O) } (static)
   val((P, (O,O)),S,1) :             schema(N,P,1), not p_static(P); % { P(O) }
   val((P,(O,O2)),S,1) : object(O2), schema(N,P,2), not p_static(P); % { P(O,O2) : O2 }
   val((P,(O2,O)),S,1) : object(O2), schema(N,P,3), not p_static(P)  % { P(O2,O) : O2 }
} = 1 :- inv(N), inv_non_empty(N), object(O), node(S).
```

Figure 3: Fragments of ASP code for different extensions of the basic ASP code.

(Blocksworld and Gripper, 3 labels) and 37,800 (Grid, 4 labels), which are not done exhaustively. In the ASP approach, ASP(G) is called a maximum number of times that is given by a bound on the number of objects, and in many cases, the models found are shown to be optimal. Among the extensions of the base code that have the greatest impact on performance, there are two: learning invariants while forcing dynamic predicates to appear in an invariant, and calling the solver separately for each number of objects as opposed to letting the solver search for this value.

6.1 Examples of Learned Representations

As in the work of Bonet and Geffner (2020), the learned representations are often more succinct than the hand-crafted representations, usually using the same predicate to represent different relations. For example, in Hanoi-3x3, the obtained model has the single action schema:

hanoi-3x3

```
MOVE(d,to,from):
 Static: NEQ(d,to), NEQ(d,from), NEQ(to,from), -BIGGER(d,to)
 Pre: -p(to,d), -p(from,from), p(d,d), p(to,to), p(from,d)
 Eff: -p(to,to), -p(from,d), p(to,d), p(from,from)
```

The predicate $p(x_1, x_2)$ denotes $on(x_2, x_1)$ if $x_1 \neq x_2$, and $clear(x_1)$ otherwise. The solver synthesizes the *exactly-1* invariant scheme $\{p(o, x) : x\}$ for each object o which says that each disc must be clear or have another disc above it.

For Gripper-3, the solver learns the schemas:

gripper-3

```
MOVE(x,to,from)
  Static: NEQ(x,from), NEQ(to,from), -B1(x,to), -B2(x,x), B1(from,x)
  Pre: -Nat(from), Nat(x), Nat(to)
  Eff: -Nat(to), Nat(from)
DROP(g,b,r)
  Static: NEQ(g,r), -B2(r,b), B1(g,g)
  Pre: -Nat(r), -at-hold(r,b), Nfree(g), at-hold(g,b)
  Eff: -Nfree(g), -at-hold(g,b), at-hold(r,b)
PICK(g,r,b)
  Static: NEQ(g,r), -B2(r,b), B1(g,g)
  Pre: -Nat(r), -Nfree(g), -at-hold(g,b), at-hold(r,b)
  Eff: -at-hold(r,b), Nfree(g), at-hold(g,b)
```

Here, $at\text{-}hold(x, ball)$ represents $at(x, ball)$ if x is a room, and $hold(x, ball)$ if x is a gripper. The binary static predicate $B_1(x_1, x_2)$ encodes that x_1 is a gripper if $x_1 = x_2$, or that x_1 and x_2 are rooms otherwise, and $B_2(x_1, x_2)$ encodes pairs of objects (x_1, x_2) that are not a room and a ball.

The schemas learned for Blocks-2 are:

blocks2-4

```
NEWTOWER(x1,x2)
  Static: NEQ(x1,x2)
  Pre: -clear(x1), -p(x1,x1), clear(x2), p(x2,x1)
  Eff: -clear(x2), -p(x2,x1), p(x1,x1)
STACK(x1,x2)
  Static: NEQ(x1,x2)
  Pre: -clear(x1), -clear(x2), -p(x2,x1), p(x1,x1)
  Eff: -p(x1,x1), clear(x2), p(x2,x1)
MOVE(x1,x2,x3)
  Static: NEQ(x1,x2), NEQ(x1,x3), NEQ(x2,x3)
  Pre: -clear(x1), -clear(x3), -p(x3,x1), clear(x2), p(x2,x1)
  Eff: -clear(x2), -p(x2,x1), clear(x3), p(x3,x1)
```

In these schemas, the predicate $p(x_1, x_2)$ represents

$on(x_2, x_1)$ or $ontable(x1)$ whether $x_1 \neq x_2$ or not. The solver finds the invariant $\{p(x, o) : o\}$ that says that a block is either on top of another, or on the table.

7 Partial and Noisy Inputs

A key assumption about the proposed learning paradigm, borrowed from Bonet and Geffner (2020), is that the input graph is **complete** (no missing nodes or edges) and **noise free** (different nodes stand for different states). We show next how to relax these assumptions.

7.1 Partial Graphs

The robustness to incomplete information is analyzed by feeding the learner with a **partial graph** G' of a true but hidden complete graph G. The partial graph G' contains some of the nodes n from G, some of the edges among these nodes from G, and counts of the number of outgoing edges (n, n') in G that are not necessarily in G', along with their labels.

The **learning task** with a **partial graph** G' is set to the following: find a planning problem $P = \langle D, I\rangle$ and a mapping from the nodes n in G' to unique states $s(n)$ over P, and a mapping from the edges (n, n') in G' to ground actions $a(n, n')$ such that 1) action $a(n, n')$ is applicable in the state $s(n)$, 2) the state $s(n')$ is the result of applying action $a(n, n')$ in the state $s(n)$, and 3) the set of actions in P applicable in each state $s(n)$ match the outgoing edge counts for n (i.e., there are k outgoing edges from n with label L iff there are k applicable actions in $s(n)$ with the same label). If the partial graph G' is the complete graph G, the learning task is equivalent to the learning task defined in Section 3.1.

In the experiments, we consider partial graphs $G' = G_p$ obtained by performing a random walk in G until p per cent of the *edges* in G have been traversed. The sampled graphs G_p represent information that an agent can acquire in a real environment where different states can be distinguished.

The partial graphs G_p are represented in the programs as the complete graphs G; i.e., using the atoms `node(S)` and `tlabel((S1,S2),L)` for the nodes and the labeled edges in G_p, respectively. In addition, the representation of G_p is extended with the atoms `tlabel((S1,S2),L)` and `unvisited(S1,S2)` for every edge `(S1,S2)` such that node `S1` is visited but the edge `(S1,S2)` is not. Since, as we will see, the program will make not use of the identity of the node `S2` in these edges, the atoms `tlabel((S1,S2),L)` for unvisited edges provides just a way for encoding the outgoing node counts. Indeed, the atoms `tlabel((S1,S2),L)` are used to activate the rules in lines 47–51 of the base code in Fig. 2 and guarantee that the outgoing edges of `S1` have a corresponding ground action applicable, while the `unvisited(S1,S2)` atoms are used to block the rules in lines 53–56 that relate the literals in the states associated with the nodes `S1` and `S2`. Every solution for the complete graph G is a solution of the partial graph G_p but not the other way around, as G_p may admit other models.

Experimental results about the performance of the learner over sampled partial graphs G_p are shown in Table 3. For each planning instance P considered that produced a model that generalizes, we sample 10 partial graphs G_p from $G = G(P)$, for each value of p in $\{20, 40, 60, 80\}$. The table shows the percentage p of edges sampled, the number of objects found, the number n of the 10 runs where a model was found, and the number of runs where the best model verified (v) and was shown to be optimal (opt). Since the most common reason that partial graphs G_p with a small number of sampled edges p lead to action schemas that do not generalize is that they are too simple, we also report the cost vector $V(M)$ for the best and worst models found in the 10 runs. In these vectors (Section 5.1), the three first entries are the sum of action arities, the sum of dynamic predicate arities, and the sum of static predicate arities. It can be seen that in most domains, the models become more complex (costly) as the number of sampled edges p is increased. The exception is Blocks-2 where the ten models found verify for almost all values of p, and they all have the same cost vectors. The models that generalize in Blocks-2 are found for $p = 20$ (20% of the edges sampled). In most domains, good generalization results are obtained for $p = 60$.

7.2 Noisy Edges and States

We also tested robustness by labeling certain edges (n, n') in the input graph as noisy, with the result that the states $s(n')$ represented by the nodes n' in such edges are **ambiguous** and cannot be assumed to be different than the other states. We can think of the nodes as represented by images, for example, such that it's not clear if a given image is the same as an image already seen or not. It is up to the solver to resolve these ambiguities. As before, the number of edges (n, n') in the input graph that are considered noisy is determined by a sampling parameter q, that ranges between 1 and 100, such that q represents the percentage of edges that are **not** noisy, so that $100 - q$ is the percentage of noisy edges. In the experiments, the graph is assumed to be complete, but the formulation can combine noisy edges and partial graphs.

The graph G_q that is fed to the solver is the result of extending a partial or complete graph G with *ambiguous* nodes that represent the states reached after traversing a noisy edge. More in detail, the labeled edges $((n, n'), l)$ that are noisy are replaced by new labeled edges $((n, n^*), l)$ where n^* is a new ambiguous node unique to the pair (n, n'). To account for the uncertainty associated to them, we allow their corresponding states to be the same as those of other (ambiguous or non-ambiguous) nodes. Next, for every labeled edge $((n', n''), l')$ in the resulting graph, we add the new labeled edge $((n^*, n''), l')$. This allows the ambiguous node n^* to access the nodes that are reachable from n', some of which may be also ambiguous. The number of nodes in G_q can thus be larger than the number of nodes in G, with the excess nodes being multiple copies of the noisy nodes.

The ASP encoding of this extension introduces the facts `ambiguous(n)` for every ambiguous node `n` in G_q, and extends the constraint in line 59 in Fig. 2 with the literals `not ambiguous(S1)` and `not ambiguous(S2)` to allow ambiguous nodes to represent the same states as other nodes. As with partial graphs, the solutions of the noisy graph G_q are a superset of the solutions of the noise-free graph G.

Table 4 shows results on complete graphs for different percentages q of **noise-free** edges. From the table, we ob-

Domain	#obj	p	n	v	*opt*	$\min V(M)$	$\max V(M)$
blocks2-4	4	20	10	10	3	(7, 2+3, 0)	(7, 2+3, 0)
blocks2-4	4	40	10	10	2	(7, 2+3, 0)	(7, 2+3, 0)
blocks2-4	4	60	10	9	1	(7, 2+3, 0)	(7, 2+3, 0)
blocks2-4	4	80	10	10	0	(7, 2+3, 0)	(7, 3+3, 0)
blocks2-4	4	100	10	10	0	(7, 2+3, 0)	(7, 3+3, 0)
grid-v0-3x4	4	20	10	0	10	(6, 2+2, 2)	(8, 1+1, 4)
grid-v0-3x4	4	40	10	6	10	(7, 2+2, 2)	(8, 2+2, 2)
grid-v0-3x4	4	60	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v0-3x4	4	80	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v0-3x4	4	100	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v1-3x4	4	20	10	2	10	(4, 1+1, 2)	(4, 2+2, 2)
grid-v1-3x4	4	40	10	5	10	(4, 2+2, 2)	(4, 2+2, 2)
grid-v1-3x4	4	60	10	9	10	(4, 2+2, 2)	(4, 2+2, 2)
grid-v1-3x4	4	80	10	5	10	(4, 2+2, 2)	(4, 2+2, 2)
grid-v1-3x4	4	100	10	9	10	(4, 2+2, 2)	(4, 2+2, 2)
gripper-2	4	20	10	0	9	(5, 3+3, 2)	(7, 2+2, 2)
gripper-2	4	40	10	1	0	(6, 3+4, 3)	(8, 3+3, 3)
gripper-2	4	60	10	4	0	(8, 1+2, 1)	(9, 3+5, 4)
gripper-2	4	80	10	8	0	(8, 2+3, 2)	(8, 3+5, 4)
gripper-2	4	100	10	6	0	(8, 3+4, 1)	(9, 3+5, 3)
gripper-3	4	20	2	1	0	(8, 2+3, 4)	(9, 3+5, 4)
gripper-3	4	40	1	1	0	(9, 3+5, 4)	(9, 3+5, 4)
gripper-3	4	60	1	0	0	(9, 3+4, 3)	(9, 3+4, 3)
gripper-3	4	80	3	2	0	(8, 3+5, 3)	(9, 3+5, 3)
gripper-3	4	100	1	1	0	(9, 3+5, 3)	(9, 3+5, 3)
hanoi-3x3	6	20	10	0	5	(2, 1+1, 2)	(3, 1+1, 2)
hanoi-3x3	6	40	10	6	0	(3, 1+2, 2)	(3, 1+2, 2)
hanoi-3x3	6	60	10	10	0	(3, 1+2, 2)	(3, 1+2, 2)
hanoi-3x3	6	80	10	10	0	(3, 1+2, 2)	(3, 2+3, 2)
hanoi-3x3	6	100	9	9	0	(3, 1+2, 2)	(3, 2+3, 2)

Table 3: Learning from partial graphs G_p where p is the percentage of sample edges from $G(P)$ and P is the instance shown on the left. Each experiment is run 10 times, and min and max cost vectors $V(M)$ of best models found are shown (Section 5). Results over complete graphs vary over the 10 runs because of the nondeterminism involved in calling CLINGO with multiple threads. Column n shows how many solutions were found in the 10 runs, v how many of them verify (generalize to larger instances), and *opt* how many were proved optimal. For reference, the cost vectors of the **intended models** for these domains are (7, 2+3, 0) for Blocks-2, (8,2+2,4) for Grid-v0, (4, 2+2, 4) for Grid-v1, (8, 2+3, 4) for Gripper, and (3, 1+2, 2) for Hanoi.

Domain	#obj	q	n	v	*opt*	$\min V(M)$	$\max V(M)$
blocks2-4	4	20	2	2	0	(7, 3+5, 2)	(7, 3+5, 2)
blocks2-4	4	40	6	6	0	(7, 3+5, 0)	(7, 3+5, 0)
blocks2-4	4	60	9	6	0	(7, 3+4, 0)	(7, 3+4, 0)
blocks2-4	4	80	9	9	0	(7, 2+3, 0)	(7, 2+3, 0)
blocks2-4	4	100	10	10	0	(7, 2+3, 0)	(7, 2+3, 0)
grid-v0-3x4	4	20	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v0-3x4	4	40	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v0-3x4	4	60	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v0-3x4	4	80	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v0-3x4	4	100	10	10	10	(8, 2+2, 2)	(8, 2+2, 2)
grid-v1-3x4	4	20	10	6	10	(4, 2+2, 2)	(4, 2+2, 2)
grid-v1-3x4	4	40	10	7	10	(4, 2+2, 2)	(4, 2+2, 2)
grid-v1-3x4	4	60	10	5	10	(4, 2+2, 2)	(4, 2+2, 2)
grid-v1-3x4	4	80	10	6	10	(4, 2+2, 2)	(4, 2+2, 2)
grid-v1-3x4	4	100	10	9	10	(4, 2+2, 2)	(4, 2+2, 2)
gripper-2	4	20	10	5	0	(8, 3+4, 1)	(8, 3+4, 1)
gripper-2	4	40	10	9	0	(8, 2+3, 4)	(8, 2+3, 4)
gripper-2	4	60	10	7	0	(8, 2+3, 1)	(8, 2+3, 1)
gripper-2	4	80	10	5	0	(8, 2+3, 1)	(8, 2+3, 1)
gripper-2	4	100	10	6	0	(8, 3+4, 1)	(8, 3+4, 1)
gripper-3	4	20	0	0	0	***	***
gripper-3	4	40	0	0	0	***	***
gripper-3	4	60	0	0	0	***	***
gripper-3	4	80	4	4	0	(8, 3+4, 3)	(8, 3+4, 3)
gripper-3	4	100	1	1	0	(9, 3+5, 3)	(9, 3+5, 3)
hanoi-3x3	6	20	0	0	0	***	***
hanoi-3x3	6	40	1	1	0	(3, 1+2, 2)	(3, 1+2, 2)
hanoi-3x3	6	60	6	6	0	(3, 1+2, 2)	(3, 1+2, 2)
hanoi-3x3	6	80	4	4	0	(3, 1+2, 2)	(3, 1+2, 2)
hanoi-3x3	6	100	9	9	0	(3, 1+2, 2)	(3, 1+2, 2)

Table 4: Learning with noisy edges and states. Results shown on complete graphs with a percentage q of edges that are not noisy over 10 runs, along with minimum and maximum cost vectors $V(M)$ of best models found. The column n shows how many solutions were found in these 10 runs, v how many of them verify, and *opt* how many were proved optimal. Asterisks indicate that no model was found in the 10 runs.

serve that in regular and simple models like grid-v0 and grid-v1, the approach is able to learn models that generalize even when there are many noisy edges. For more complex domains, as the parameter q decreases the models that verify tend to decrease. The ambiguity resulting from the "noise" is not always detrimental to performance, however. In Gripper-2, for example, for $q = 40$, 9 out of the 10 runs yield models that generalize, while for $q = 100$, only 6 of them do. In Gripper-3, with $q = 80$, 4 out of the 10 runs yield models that generalize, while for $q = 100$, only 1 does.

8 Conclusion

We have explored variations and extensions of the approach to representation learning for planning proposed by Bonet and Geffner (2020) that showed how crisp and meaningful symbolic representations can be learned from flat state spaces associated with small instances, in a way that the resulting first-order domains (action schemas and predicates) generalize to larger instances. The new ASP implementation has been shown to be more scalable and robust, as it opens new possibilities for modeling and solving the learning problem, and a higher level of abstraction to explore them. The performance improvements are significant, as the consideration of thousands of propositional SAT theories encoding all possible values of the hyperparameters, have been replaced by a simple meta-search on the number of objects, as all other decisions are left to the solver, which in many cases manages to find solutions and prove them optimal (simplest). In the new, high level ASP encoding, it has also been simple to relax some of the assumptions made in previous work, namely that the input graphs are complete and noise-free. It remains as an interesting challenge to improve the experimental results even further so that the resulting methods can be applied to learn representations, for example, from arbitrary IPC planning domains.

Acknowledgements

The work is partially supported by an ERC Advanced Grant (No 885107), by project TAILOR, funded by an EU Horizon 2020 Grant (No 952215), and by the Knut and Alice Wallenberg (KAW) Foundation under the WASP program. Hector Geffner is a Wallenberg Guest Professor at Linköping University, Sweden.

References

Aineto, D.; Jiménez, S.; Onaindia, E.; and Ramírez, M. 2019. Model recognition as planning. In *Proc. ICAPS*, 13–21.

Arora, A.; Fiorino, H.; Pellier, D.; Métivier, M.; and Pesty, S. 2018. A review of learning planning action models. *The Knowledge Engineering Review* 33.

Asai, M. 2019. Unsupervised grounding of plannable first-order logic representation from images. In *Proc. ICAPS*, 583–591.

Bajpai, A.; Garg, S.; and Mausam. 2018. Transfer of deep reactive policies for MDP planning. In *Proc. NeurIPS*, 10988–10998.

Belle, V., and Levesque, H. J. 2016. Foundations for generalized planning in unbounded stochastic domains. In *KR*, 380–389.

Bonet, B., and Geffner, H. 2020. Learning first-order symbolic representations for planning from the structure of the state space. In *Proc. ECAI*, 2322–2329.

Bonet, B.; Francès, G.; and Geffner, H. 2019. Learning features and abstract actions for computing generalized plans. In *Proc. AAAI*, 2703–2710.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. ICAPS-09*, 34–41.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Comm. ACM* 54(12):92–103.

Bueno, T. P.; de Barros, L. N.; Mauá, D. D.; and Sanner, S. 2019. Deep reactive policies for planning in stochastic nonlinear domains. In *AAAI*, volume 33, 7530–7537.

Chevalier-Boisvert, M.; Bahdanau, D.; Lahlou, S.; Willems, L.; Saharia, C.; Nguyen, T. H.; and Bengio, Y. 2019. Babyai: A platform to study the sample efficiency of grounded language learning. In *ICLR*.

Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review* 28(2):195–213.

Darwiche, A. 2018. Human-level intelligence or animal-like abilities? *Communications of the ACM* 61(10):56–67.

Diuk, C.; Cohen, A.; and Littman, M. L. 2008. An object-oriented representation for efficient reinforcement learning. In *Proc. ICML*, 240–247.

Fern, A.; Yoon, S.; and Givan, R. 2003. Approximate policy iteration with a policy language bias. In *Proc. NIPS*, 847–854.

Francès, G.; Bonet, B.; and Geffner, H. 2021. Learning general policies from small examples without supervision. In *Proc. AAAI*.

François-Lavet, V.; Bengio, Y.; Precup, D.; and Pineau, J. 2019. Combined reinforcement learning via abstract representations. In *Proc. AAAI*, volume 33, 3582–3589.

Garnelo, M., and Shanahan, M. 2019. Reconciling deep learning with symbolic artificial intelligence: representing objects and relations. *Current Opinion in Behavioral Sciences* 29:17–23.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. Answer set solving in practice. *Synthesis lectures on artificial intelligence and machine learning* 6(3):1–238.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming* 19(1):27–82.

Geffner, H. 2018. Model-free, model-based, and general intelligence. In *Proc. IJCAI*, 10–17.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning generalized reactive policies using deep neural networks. In *Proc. ICAPS*, 408–416.

Hafner, D.; Lillicrap, T.; Fischer, I.; Villegas, R.; Ha, D.; Lee, H.; and Davidson, J. 2019. Learning latent dynamics for planning from pixels. In *Proc. ICML*, 2555–2565.

Hafner, D.; Lillicrap, T.; Norouzi, M.; and Ba, J. 2020. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Morgan & Claypool.

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI*, 918–923.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training deep reactive policies for probabilistic planning problems. In *Proc. ICAPS*, 422–430.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* 113(1-2):125–148.

Kingma, D. P., and Welling, M. 2014. Auto-encoding variational bayes. In *ICLR*.

Lake, B. M.; Ullman, T. D.; Tenenbaum, J. B.; and Gershman, S. J. 2017. Building machines that learn and think like people. *Behavioral and Brain Sciences* 40.

Lifschitz, V. 2019. *Answer set programming*. Springer.

Marcus, G. 2018. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*.

Martín, M., and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Applied Intelligence* 20(1):9–19.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *Artificial Intelligence Magazine* 21(2):35–56.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

Pearl, J. 2018. Theoretical impediments to machine learning with seven sparks from the causal revolution. *arXiv preprint arXiv:1801.04016*.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588(7839):604–609.

Shanahan, M.; Nikiforou, K.; Creswell, A.; Kaplanis, C.; Barrett, D.; and Garnelo, M. 2020. An explicitly relational neural network architecture. In *Proc. ICML*, 8593–8603.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *Proc. AAAI*, 991–997.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In *Proc. AAAI*, 6294–6301.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence* 171(2-3):107–143.