Learning First-Order Symbolic Planning Representations That Are Grounded

Andrés Occhipinti,[1] Blai Bonet, [1] Hector Geffner [1,2,3]

[1] Universitat Pompeu Fabra, Barcelona, Spain
[2] Institució Catalana de Recerca i Estudis Avançats (ICREA), Barcelona, Spain
[3] Linköping University, Linköping, Sweden
andres.occhipinti@upf.edu, bonetblai@gmail.com, hector.geffner@upf.edu

Abstract

Two main approaches have been developed for learning first-order planning (action) models from unstructured data: combinatorial approaches that yield crisp action schemas from the structure of the state space, and deep learning approaches that produce action schemas from states represented by images. A benefit of the former approach is that the learned action schemas are similar to those that can be written by hand; a benefit of the latter is that the learned representations (predicates) are grounded on the images, and as a result, new instances can be given in terms of images. In this work, we develop a new formulation for learning crisp first-order planning models that are grounded on *parsed images,* a step to combine the benefits of the two approaches. Parsed images are assumed to be given in a simple O2D language (objects in 2D) that involves a small number of unary and binary predicates like 'left', 'above', 'shape', etc. After learning, new planning instances can be given in terms of pairs of parsed images, one for the initial situation and the other for the goal. Learning and planning experiments are reported for several domains including Blocks, Sokoban, IPC Grid, and Hanoi.

Introduction

One of the key open problems in AI is how to combine learning and reasoning, in particular when the learning data is not structured for reasoning. In planning, there are effective reasoning mechanisms for planning but they rely on models comprised of predicates and action schemas which are usually provided by hand. A number of proposals have been advanced for learning and refining these models, but most assume that the domain predicates are known (Yang, Wu, and Jiang 2007; Zhuo et al. 2010; Mourao et al. 2012; Zhuo and Kambhampati 2013; Aineto, Celorrio, and Onaindia 2019; Lamanna et al. 2021). The problem of learning the domain predicates and the action schemas at the same time is more challenging. A clever approach for addressing this problem given sequences of grounded actions was developed in the LOCM system (Cresswell, McCluskey, and West 2013; Gregory and Lindsay 2016), although the approach is heuristic and incomplete. Two recent formulations have addressed the problem more systematically and without assuming that action arguments are observable. One is a combinatorial approach that yields crisp action schemas from the structure of the state space (Bonet and Geffner 2020; Rodriguez et al. 2021); the other is a deep learning approach that produces action schemas from states represented by images (Asai 2019). A benefit of the combinatorial approach is that it accommodates and exploits a natural inductive bias where fewer, simpler action schemas and predicates are preferred; a bias that results in learned action schemas that are similar to those that can be written by hand. A benefit of deep learning approaches is that the learned representations (predicates) are grounded on the images, and as a result, new instances can be given in terms of them.

The aim of this work is to develop a new formulation for learning crisp first-order planning models that are grounded on *parsed images*, a step in the way to combine the benefits of the combinatorial and deep learning approaches. Parsed images are assumed to be given in a simple O2D language, for "objects in 2D", that involves a small number of unary and binary "visual" predicates like 'left', 'above', 'shape', etc. The learning problem becomes the problem of learning the lifted domain (action schemas and domain predicates) along with the *grounding* of the learned domain predicates so that they can be evaluated on any parsed image.[1] A number of vision modules can be used to map images into the parsed representations (Redmon et al. 2016; Redmon and Farhadi 2017; Locatello et al. 2020), but this is outside the scope of this work. After learning, new planning instances can be given in terms of pairs of parsed images, one corresponding to the initial situation and the other to the goal, and such instances can be solved with any off-the-shelf planner and may involve many more objects than those used in training. Learning and planning results for several domains are reported, including Blocks, Towers of Hanoi, the n-sliding-puzzle, IPC Grid, and Sokoban.

The paper is organized as follows. We discuss related work, review the basics of classical planning, and introduce the O2D language and the learning formulation. An implementation of the learning formulation as an answer set program is then sketched (full details in the appendix), and experimental results are presented and analyzed.

[1] Grounding a predicate (symbol) means to provide a semantics for it in the form of a denotation function, and should not be confused with grounding of the action schemas. For the problem of grounding symbols in the "real world", see Harnad [1990].

Related Work

Most works on learning action schemas from traces assume that the domain predicates are known (Yang, Wu, and Jiang 2007; Walsh and Littman 2008; Zhuo et al. 2010; Mourao et al. 2012; Zhuo and Kambhampati 2013; Stern and Juba 2017; Aineto, Celorrio, and Onaindia 2019; Lamanna et al. 2021). The problem of learning the action schemas and the predicates at the same time is more challenging as the structure of the states is not available at all. The LOCM systems (Cresswell, McCluskey, and West 2013; Cresswell and Gregory 2011; Gregory and Lindsay 2016; Lindsay 2021) addressed this problem assuming input traces that feature sequences of ground actions. The inference of action schemas and predicates follows a number of heuristic rules that manage to learn challenging planning domains but whose soundness and completeness properties have not been studied. A general formulation of the learning problem from complete input traces that feature just action names and black-box states is given by Bonet and Geffner [2020], and extensions for dealing with incomplete and noisy traces by Rodriguez et al. [2021] (see also Verma, Marpally, and Srivastava [2021]). An alternative, deep learning approach for learning action schemas and predicates from states represented by images is advanced by Asai [2019]. The advantages of a deep learning approach based on images are several: it does not face the scalability bottleneck of combinatorial approaches, it is robust to noise, and it yields representations *grounded* in the images. The limitation is that the learned planning domains tend to be complex and opaque. For example, Asai reports 518,400 actions for a Blocksworld instance with 3 blocks. Some recent approaches learn more compact, rule-based representations from images without resorting to deep learning. One such approach is that of (Xie et al. 2022) in which so-called visual rewrite rules are learned, which are action-dependent graphical rules that describe action effects as local visual changes around the agent. Methods for learning *propositional* planning representations that are grounded have also been proposed (Konidaris, Kaelbling, and Lozano-Perez 2018; Asai and Fukunaga 2018; Asai and Muise 2020) but they are bound to work in a single state space involving a fixed set of objects.

Classical Planning

A (classical) planning instance is a pair $P = \langle D, I \rangle$ where D is a first-order planning domain and I represents instance information (Geffner and Bonet 2013; Ghallab, Nau, and Traverso 2016; Haslum et al. 2019). The planning domain D contains a set of predicates (predicate symbols) p 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 instance information is a tuple $I = \langle O, Init, Goal \rangle$ where O is a (finite) set of objects (object names) o_i, and $Init$ and $Goal$ are sets of ground atoms $p(o_1, \ldots, o_k)$ or their negations, with $Init$ being consistent and complete; i.e., for each ground atom $p(o_1, \ldots, o_k)$, either the atom or its negation is (true) in $Init$. The set of all ground atoms in $P = \langle D, I \rangle$, $At(P)$, is given by all the atoms that can be formed from the predicates in D and the objects in I, while the set of ground actions $A(P)$ is given by the action schemas with their arguments replaced by objects in P. A state s over P is a maximally consistent set of ground literals representing a truth valuation over the atoms in $At(P)$, and a ground action $a \in A(P)$ is applicable in s, written $a \in A(s)$ when its preconditions are (true) in s. A state s' is the successor of ground action a in state s, written $s' = f(a, s)$ for $a \in s$ if the effects of a are true in s' and the truth of atoms not affected by a is the same in s and s'. Finally, an action sequence $a_0, \ldots, a_n$ is a plan for P if there is a state sequence $s_0, \ldots, s_{n+1}$ such that s_0 satisfies $Init$, s_{n+1} satisfies $Goal$, $a_i \in A(a_i)$, and $s_{i+1} = f(a_i, s_i)$.

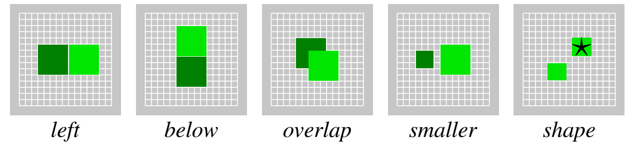


Figure 1: Depiction of the five binary relations in O2D.

Language of Parsed Images: O2D

Object-recognition vision systems typically map images into object-tuples of the form $\{\langle type(c), loc(c), bb(c), att(c) \rangle\}_c$ that encode the different objects c in the scene, their type or class, their location and bounding box coordinates, and some visual attributes like color or shape (Redmon et al. 2016; Redmon and Farhadi 2017; Locatello et al. 2020). We use a similar encoding of scenes but rather than representing the exact locations of objects, spatial relations are represented *qualitatively* (Cohn and Renz 2008). More precisely, a scene is represented by a set of ground atoms over a language that we call O2D for *Objects in 2D space*. O2D is a first-order language with signature $\Sigma = (C, U, R)$ where C stands for a set of constant symbols representing objects and shapes, U stands for a set of unary predicates, and R stands for a set of binary predicates. The unary predicates in U stand for visually different object types and characteristics, while the binary predicates are $R = \{left, below, overlap, smaller, shape\}$, representing if one object is right to the left of or right below another object, if two objects overlap, if one object is smaller than another, and the shape of an object; see Figure 1.

A scene is represented in O2D as a set of *ground atoms* over the symbols in $\Sigma = (C, U, R)$. We refer to scene representations in O2D as *O2D states*. Scenes and their corresponding O2D states for Blocks-world, Tower-of-Hanoi, and Sokoban are shown in Figure 2, with renderings obtained with PDDLGym (Silver and Chitnis 2020).

Groundings

A grounded predicate q is a predicate that can be evaluated in any O2D state s; i.e., if o is a tuple of objects in s of the same arity as q, then $q(o)$ is known to be true or to be false in s.

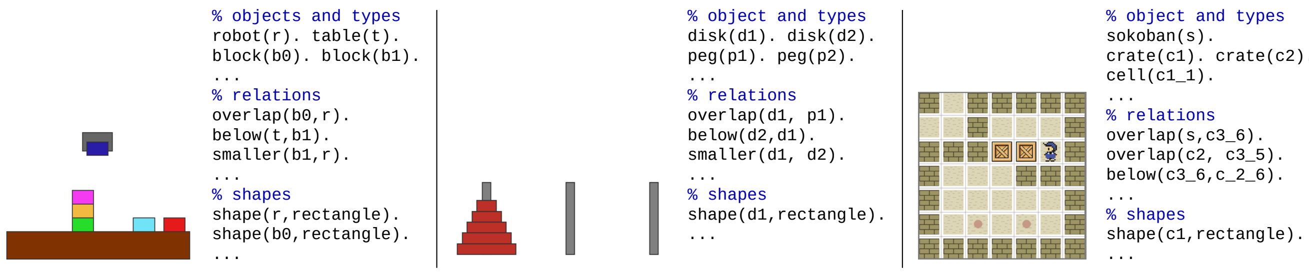


Figure 2: Scenes for three of the domains considered and their O2D representations (see Appendix for details).

The predicates p appearing in a planning domain D are grounded by assuming a pool $\mathcal{P}$ of grounded predicates and a *grounding function* σ that maps the domain predicates p into grounded predicates $q = \sigma(p)$ in the pool with the same arity as p. The result is a *grounded domain*:

Definition 1 (Grounded Domain)**.** *A grounded domain over a pool of grounded predicates $\mathcal{P}$ is a pair $\langle D, \sigma \rangle$ where D is a planning domain D and σ is a function that maps each predicate p in D into a predicate $q = \sigma(p)$ in $\mathcal{P}$.*

The truth value of an atom $p(o)$ in a scene s is the value of the atom $q(o)$ when q is the grounding of p: i.e., when $\sigma(p) = q$. The way in which the pool of grounded predicates $\mathcal{P}$ is constructed is similar to the way in which a pool of unary predicates is defined by Bonet, Frances, and Geffner [2019] for generating Boolean and numerical features: there is a set of *primitive predicates* and a set of description logic grammar rules (Baader, Horrocks, and Sattler 2008) for defining new compound predicates from them. The differences are that $\mathcal{P}$ contains nullary and binary predicates as well, and that the primitive predicates are not the domain predicates, that are to be learned, but the O2D predicates that are known and grounded. The denotation of the predicates defined by a grammar rule is determined by the semantics of the rule and the denotation of predicates appearing in the right hand side. The actual description logic (DL) grammar considered for unary predicates (concepts) is

$$C \leftarrow U \mid \top \mid \bot \mid \exists R.C \mid C \sqcap C'$$

meaning that unary O2D predicates (U), the universal true/false predicates, existential restrictions, and intersections (conjunctions) are all unary predicates. The rules for binary predicates (roles) are:

$$R \leftarrow R \mid R^{-1} \mid R \circ R'$$

meaning that binary O2D predicates (R_0), inverses, and role compositions are binary predicates. Finally, nullary predicates are obtained from unary predicates C and C' as $C \sqsubseteq C'$, an expression that is true in states where the extension of C is a subset of the extension of C'.

The set of predicates of complexity no greater than i is denoted as $\mathcal{P}_i$, where the complexity of top and bottom is 0, the complexity of the O2D predicates is 1, and the complexity of derived predicates is 1 plus the sum of the complexities of the predicate involved in the rule. For a given pool of O2D states, the sequence $\mathcal{P}_0, \ldots, \mathcal{P}_m$ is constructed iteratively, pruning duplicate predicates (predicates with the same denotation). The pool $\mathcal{P}$ is $\mathcal{P}_m$ for some bound $m > 0$.

Example. In Blocks, the atom *clear*(b) is true when no block is above b (not *some_above*(b)), and b is not held by the robot (not $holding(b)$). Given the O2D representation of Blocks in Fig. 2, *some_above* and *holding* can be grounded to the derived O2D predicates $\sigma(\textit{some\_above}) = \exists\, below.block$ and $\sigma(\textit{holding}) = \exists\, overlap.robot$, both of complexity 2.

Learning: Formulation

The training data $\mathcal{D}$ for learning grounded domains is $\mathcal{D} = \langle \mathcal{T}, \mathcal{S}, L, F \rangle$, where $\mathcal{T}$ and $\mathcal{S}$ are sets of O2D states (scene representations) over one or more instances, $\mathcal{T} \subseteq \mathcal{S}$; L is a set of action labels α (schema names), and $F_\alpha(s)$ is the multiset of O2D states s' that follow s in $\mathcal{T}$ when an action with label $\alpha \in L$ is performed.

Such states s' are part of $\mathcal{S}$ but not necessarily of $\mathcal{T}$ that is a subset of $\mathcal{S}$. In the formulation of Bonet and Geffner [2020], the states in the data are black-boxes, not O2D states, and $\mathcal{T} = \mathcal{S}$.

The grounded domain $\langle D, \sigma \rangle$ to be learned from this input contains one action schema per action label α, and determines a function $h = h^D_\sigma$ that maps arbitrary O2D states s into planning states $h(s)$ over D with the same set of objects. More precisely, $h(s)$ is the truth valuation over the atoms $p(o)$, where p is a predicate in D and o is a *tuple of objects* from s of the same arity as p, given by set of literals:

$$\begin{aligned} h(s) \stackrel{\text{def}}{=} \; & \{\, p(o) \mid q^s(o) = 1, \sigma(p) = q, p \text{ in } D, o \text{ in } s \,\} \cup \\ & \{\, \neg p(o) \mid q^s(o) = 0, \sigma(p) = q, p \text{ in } D, o \text{ in } s \,\} \end{aligned}$$

which has positive literals $p(o)$ for $\sigma(p) = q$ and $q(o)$ true in s, and negative literals $\neg p(o)$ for $\sigma(p) = q$ and $q(o)$ false in s.

For action schema α in D, and planning state $\bar{s}$, let $F^D_\alpha(\bar{s})$ represent the multiset formed by the states $\bar{s}'$ that follow $\bar{s}$ after ground instantiations of the schema α in $\bar{s}$; i.e.,

$$F^D_\alpha(\bar{s}) \stackrel{\text{def}}{=} \{\!\{\, \bar{s}' \mid \bar{s}' = f(a, \bar{s}), a \in A(\bar{s}), label(a) = \alpha \,\}\!\}$$

where $A(\bar{s})$ is the set of ground instances of schema α over the objects in $\bar{s}$ that are applicable in $\bar{s}$, and f is the state-transition function determined by D for the ground action a in state $\bar{s}$. The learning task can be formally defined as follows:

Definition 2 (Learning Task)**.** *Let $\mathcal{D} = \langle \mathcal{T}, \mathcal{S}, L, F \rangle$ be the input data, and let $\mathcal{P}$ be a pool of grounded predicates. The **learning task** $L(\mathcal{D}, \mathcal{P})$ is to obtain a (simplest) grounded* *domain* $\langle D, \sigma\rangle$ *with one action schema per label* α *in* L *such that the resulting abstraction function* $h = h^D_\sigma$ *complies with the following two constraints:*

C1. If $s \neq s'$*, then* $h(s) \neq h(s')$*, for* $s, s' \in \mathcal{T}$*; and*
C2. $F^D_\alpha(h(s)) = \{\{h(s') \mid s' \in F_\alpha(s)\}\}$ *for* $s \in \mathcal{T}$, $\alpha \in L$.

The first constraint C1 says that the abstract (planning) states for different O2D states in $\mathcal{T}$ must be different, while C2 says that the abstraction function h must represent an isomorphism. Indeed, if $G_{\mathcal{D}}$ is the *data graph* with vertex set $\mathcal{S}$ and edges (s, α, s') for $s \in \mathcal{T}$, $s' \in F_\alpha(s)$ and $\alpha \in L$, and G_h is the *planning graph* with vertex set V_h equal to the planning states reachable from $\{h(s) \mid \mathcal{S}\}$ and edges $(\bar{s}', \alpha, \bar{s})$ for $\bar{s} \in V_h$, $a \in A(\bar{s})$, $label(a) = \alpha$, and $\bar{s}' \in F^D_\alpha(\bar{s})$, then:[2]

Theorem 3. *If* $\langle D, \sigma\rangle$ *is a solution of the learning task* $L(\mathcal{D}, \mathcal{P})$ *and* $\mathcal{T} = \mathcal{S}$*, the data and planning graphs* $G_{\mathcal{D}}$ *and* G_h *for* $h = h^D_\sigma$ *are isomorphic.*

The *complexity* of a domain D is defined by a lexicographic cost function that considers, in order, the arity of the action schemas, the sum of the arities for non-static predicates, the same sum for static predicates, the number of effects, and the number of preconditions. The first three criteria are from Rodriguez et al. [2021]. The complexity of a grounded domain $\langle D, \sigma\rangle$ is the complexity of D, and a grounded domain is *simplest* when it has minimal complexity. The **optimal solutions** of the learning task $L(\mathcal{D}, \mathcal{P})$ in Definition 2 are the simplest grounded domains that satisfy constraints C1 and C2.

Given a grounded domain $\langle D, \sigma\rangle$, any pair of O2D states s_0 and s_g defines a classical planning problem $P = \langle D, I\rangle$ where $I = \langle O, Init, Goal\rangle$ is such that the objects in O are the ones in s_0 and s_g, $Init = h(s_0)$, and $Goal = h(s_g)$.

Properties and Scope

Some assumptions in the formulation are 1) actions that change the planning state must change the O2D state (cf. C1), 2) the objects in the planning instances are the ones appearing in the O2D states, and 3) the target language for learning is lifted STRIPS with negation. These assumptions have concrete implications; e.g., in Sokoban, the cells in the grid must appear as O2D objects, else assumption 1 is violated. Likewise, in Sliding Tile, the tiles suffice for distinguishing O2D states, but cells as objects are needed in STRIPS.[3]

The completeness of the approach can be characterized in terms of a "hidden" domain D. Namely, if the O2D states s are mere "visualizations" of planning states $\bar{s}$ over D, and there is a function $h = h^D_\sigma$ given the pool of predicates $\mathcal{P}$ that allows us to recover the planning states $\bar{s}$ from their visualizations, then the grounded domain $\langle D, \sigma\rangle$ is a solution of the learning task $L(\mathcal{D}, \mathcal{P})$:

Theorem 4. *Let* D *be a (hidden) planning domain, let* $\mathcal{D} = \langle \mathcal{T}, \mathcal{S}, L, F\rangle$ *be a dataset, and let* g *be a 1-1 function that maps planning states* $\bar{s}$ *in* D *into O2D states* $g(\bar{s})$ *such that* $F_\alpha(g(\bar{s})) = \{\{g(\bar{s}') \mid \bar{s}' \in F^D_\alpha(\bar{s})\}\}$ *for* $g(\bar{s}) \in \mathcal{T}$ *and* α *in* L*. If there is a grounding function* σ *for the predicates in* D *over a pool* $\mathcal{P}$ *such that* $h = h^D_\sigma$ *is the right inverse of* g *on* $\mathcal{T}$ *(i.e.,* $g(h(s)) = s$ *for* $s \in \mathcal{T}$*), then* $\langle D, \sigma\rangle$ *is a solution for the learning task* $L(\mathcal{D}, \mathcal{P})$.

The key difference from approaches that learn action schemas *given* the domain predicates is that, in our formulation, the domain predicates are not given but must be invented and grounded using a pool of predicates that is obtained from the *given* O2D predicates.

Extensions and Variations

In some cases, we want an slight variation of the learning task $L(\mathcal{D}, \mathcal{P})$ where there is no need to distinguish all O2D states (constraint C1). For example, we may learn a relation $sep(s, s')$ that is true if s is a goal state and s' is not, and then limit the scope of C1 to such pairs (that need to be distinguished). In other cases, the addition of *domain constants* in the planning language can reduce the arity of action schemas (Haslum et al. 2019). The constants are easily learned from O2D states where they correspond to the denotation of grounded, unary, static predicates that single out one particular object per instance. Such objects are identified at preprocessing and explicitly marked as constants before learning the action schemas.

Learning: ASP Implementation

The learning task $L(\mathcal{D}, \mathcal{P})$ in Definition 2 can be cast as a combinatorial optimization problem $T_\beta(\mathcal{D}, \mathcal{P})$ once two hyperparameters are set in β: the max arity of actions, and the max number of predicates.

The problem $T_\beta(\mathcal{D}, \mathcal{P})$ is expressed and solved as an answer set program (Brewka, Eiter, and Truszczyński 2011; Lifschitz 2019; Gebser et al. 2012) using the CLINGO solver (Gebser et al. 2019), building on the code for learning *ungrounded* lifted STRIPS representations (Rodriguez et al. 2021). The main departures from Rodriguez et al. [2021] are: 1) there is no assumption that instances in the input data are represented as full state graphs ($\mathcal{T} = \mathcal{S}$), 2) there is no choice of the truth values of atoms $p(o)$ in the different nodes; instead a grounding $\sigma(p)$ for the domain predicates p is selected from $\mathcal{P}$ (actually, the name of the domain predicates is irrelevant and does not appear in the code); and 3) action arguments of ground actions are factorized, so that if there are actions of arity 4 and 15 objects, the $15^4 = 50,625$ ground actions are not enumerated. These changes allow us to learn domains that cannot be learned using the previous methods. Other departures are the use of O2D states in the input as opposed to black-box states, the introduction of domain constants in the planning language, and a more elaborated optimization criterion. The full ASP

[2]Proofs can be found in appendix.

[3]The problem of determining the "objects" in a scene for a given target planning language is subtle and not tied to our particular learning approach but to modeling in general. It also surfaces in deep learning approaches from images over the same target languages but then the problem does not become visible as the schemas and the objects are not transparent. A way out of this problem appears in (Bonet and Geffner 2020; Rodriguez et al. 2021) where the "objects" are "invented" along with the action schemas and predicates.

Domain (#inst.)	#obj.	#const.	$\lvert A\rvert$	$\lvert S\rvert$	#edges	Predicate pool P $\lvert P\rvert$	m	time
Blocks3ops (4)	5	2	3	590	2,414	13	2	1.41
Blocks4ops (5)	5	3	4	1,020	2,414	79	4	9.13
Hanoi1op (5)	8	1	1	363	1,074	14	2	2.02
Hanoi4ops (5)	8	1	4	363	1,074	14	2	2.03
Sliding Tile (7)	11	1	4	742	1,716	16	2	0.96
IPC Grid (19)	11	1	10	9,368	23,530	164	4	316.64
Sokoban1 (95)	22	3	8	1,936	5,042	18	2	8.54
Sokoban2 (24)	27	3	8	12,056	36,482	18	2	160.48

Table 1: Data pool. For each domain, columns show number of instances, max number of objects, number of domain constants, number of action labels, total number of states and edges across all instances, size of predicate pool P, complexity bound m, and time in seconds to generate P. Each instance consists of all states reachable from the initial state.

Domain	#iter	#inst.	#states	Learning time in seconds: solve	ground	verif.	total
Blocks3ops	5	3	20	0.05	1.92	0.84	2.97
Blocks4ops	7	3	16	0.29	23.37	29.42	53.70
Hanoi1op	4	2	7	0.06	1.53	0.44	2.16
Hanoi4ops	6	4	27	1.56	12.67	0.59	15.06
Sliding Tile	6	5	10	0.11	2.89	1.20	4.43
IPC Grid	27	12	127	693.44	3,536.23	2,404.87	6,653,03
Sokoban1	10	9	13	16.18	285.56	9.18	311.79
Sokoban2	11	8	56	7,250.67	5,314.35	165.19	12,740.43

Table 2: Learning results. For each domain, the first column shows the number of iterations of the incremental learner until optimal solutions that verify over all data in the pool are found. The others show the number of instances and states in the final set T constructed from the data pool, and the times in seconds for solving and grounding the ASP programs, for verification, and total time.

code is in the appendix.[4]

Experimental Results

We test the performance of the ASP program expressing the combinatorial optimization problem $T_\beta(D, P)$ on two versions of Blocks and Towers of Hanoi, the Sliding-Tile Puzzle, IPC Grid, and Sokoban. The pool of grounded predicates P is computed from the given O2D predicates as mentioned above, using complexity bounds $m = 2$ and $m = 4$ (details below). The max number of predicates is set to 12 and the maximum arity of actions is set to 3 except for Sokoban that is set to 4. The experiments are performed on Amazon EC2's `r5.8xlarge` instances that feature 32 Intel Xeon Platinum 8259CL CPUs @ 2.5GHz, and 256GB of RAM, and CLINGO is run with options '`-t 6 --sat-prepro=2`'.

Data generation. The data D for learning and validation is obtained from states $\bar{s}$ of planning instances $P_i = \langle D, I_i \rangle$, $i = 1, \ldots, n$ for each domain, encoded in STRIPS and *ordered* by the size of the state space. The O2D states $s = g(\bar{s})$ are obtained from the planning states $\bar{s}$ using a 1-to-1 "rendering" function g as in Theorem 4 with $F_\alpha(g(\bar{s}))$ set to $\{\{g(\bar{s}') \mid \bar{s}' \in f(a, \bar{s}), a \in A(\bar{s}), a \in \alpha(P_i)\}\}$. Characteristics of the data pool are shown in Table 1; further details can be found in the appendix (suppl. material). Sokoban1 and Sokoban2 refer to the same domain but different training instances: Sokoban2 contains fewer but much larger instances (the largest has 6,832 states).

Incremental learning. The data generated from the planning instances is used incrementally for computing an *optimal solution* $\langle D_i, \sigma_i \rangle$ of the learning task $L(D_i, P)$, $i = 0, \ldots, n$. D_0 and D_0 are empty, and D_{i+1} is equal to D_i, when the solution obtained from $L(D_i, P)$ verifies (generalizes) over all the data in D (satisfies constraints C1 and C2 in Definition 2). When not, D_{i+1} extends D_i with a set Δ of O2D states obtained from the first P_k instance where the verification fails. If constraint C1 is violated for a pair of states $\{g(\bar{s}), g(\bar{s}')\}$, Δ is set to the pair. Else, if C2 is violated for some states $g(\bar{s})$ and label α, Δ collects up to the first 10 such states. The set Δ extends $D_i = \langle T_i, S_i, L_i, F^i \rangle$ into D_{i+1} as follows, where $\alpha \in D$ stands for the known action labels (schema names), and L_{i+1} is the set of all such labels for all $i > 0$:

– $T_{i+1} = T_i \cup \Delta$,

– $S_{i+1} = S_i \cup \Delta \cup \bigcup \{F_\alpha(g(\bar{s})) \mid g(\bar{s}) \in \Delta, \alpha \in D\}$,

– $F_\alpha^{i+1} = F_\alpha^i \cup \{\langle g(\bar{s}), F_\alpha(g(\bar{s})) \rangle \mid g(\bar{s}) \in \Delta\}, \alpha \in D$.

Results. Table 2 shows the results of the incremental learner given the pool of data in Table 1.

For each domain, the columns show the number of iterations until an optimal model that verifies over all the instances in the data pool is found, the number of instances and states (in T) from the data pool used up to this point, and the times in seconds for grounding and solving the ASP program, for verification, and total time.

The learning task $L(D, P)$ for all domains admit solution with the pool $P = P_m$ for $m = 2$, except for IPC Grid and Blocks4ops where no solution exists for $m \leq 3$ and require a bound $m = 4$. In both cases, however, the solver takes less than 20 seconds in total to report lack of solutions for the bounds $m = 2$ and $m = 3$.

Some of the domains have been considered before, like Blocks3ops and Hanoi1op (Bonet and Geffner 2020; Rodriguez et al. 2021), but others, like IPC Grid and Sokoban are more challenging. The final model for IPC Grid involves 10 action schemas (6 of arity 2 and 4 of arity 3), while the one for Sokoban involves 8 action schemas (4 of arity 2 and 4 of arity 4). The max number of objects that ended up being used during training was 8 for IPC Grid and 21 for Sokoban2.

Learned Representations

In the experiments, data obtained from hidden planning instances was used for generating the training data. The original and learned domains, referred to as D_O and D_L, must agree on the number and name of the action schemas, but not in their arities or in the predicates involved. Table 3 compares D_O and D_L along dimensions reflected in the optimization criterion, and Fig. 3 shows learned schemas for IPC Grid and Sokoban. In general, the learned domains are not equal to the hidden domains, but they are close and equally meaningful and interpretable.

[4] Data and code will be made available.

Domain	Original domain D_O action arities	#pred.	Learned domain D_L action arities	#pred.	#c
Blocks3ops	(2 of 2, 3)	$(3, 1)$	(2 of 2, 3)	$(2, 0)$	1
Blocks4ops	(2 of 1, 2 of 2)	$(5, 0)$	(2 of 1, 2 of 2)	$(3, 0)$	2
Hanoi1op	(1 of 3)	$(2, 3)$	(1 of 3)	$(2, 1)$	0
Hanoi4ops	(4 of 3)	$(2, 3)$	(4 of 3)	$(2, 2)$	0
Sliding Tile	(4 of 3)	$(4, 2)$	(4 of 3)	$(2, 2)$	0
IPC Grid	(6 of 2, 4 of 4)	$(6, 7)$	(6 of 2, 4 of 3)	$(4, 4)$	1
Sokoban	(4 of 3, 4 of 5)	$(2, 4)$	(4 of 2, 4 of 4)	$(2, 2)$	1

Table 3: Comparison of original, hidden domains used to generate the data (D_O) and learned domains (D_L). The columns shown action arities, number of dynamic and static predicates (#pred), and number of constants in D_L (#c). E.g., D_O for Blocks4ops has 2 actions of arity 1 (Pickup and Putdown), 2 actions of arity 2 (Stack and Unstack), 5 dynamic predicates (ontable, on, holding, clear, and armempty), and no static predicates. The learned grounded domains for both Sokoban benchmarks are equal; only one is shown.

```
[Grid] Pickup(p, k):
pre: armempty, at(R, p), at(p, k)
eff: ¬armempty, ¬somecell(k), ¬at(p, k), ¬at(k, p)

[Sokoban] Pushdown(x, y, z, c):
static: below(z, y), below(y, x)
pre: at(Sok, x), at(c, y), ¬nempty(z)
eff: ¬nempty(x), nempty(z), at(Sok, y), at(y, Sok), ¬at(Sok, x)
     ¬at(x, Sok), ¬at(y, c), ¬at(c, y), at(c, z), at(z, c)
```

Figure 3: Two learned action schemas for Grid (top) and Sokoban (bottom). Predicates names our own; see text for their grounding.

The groundings obtained for the predicates of the different domains are interesting as well (predicates names are our own). For example, Sokoban uses '$nempty(c)$' atoms that hold when cell c has either a crate or the sokoban, and '$at(x, y)$' atoms that hold when object x is at y; the first is grounded on the derived O2D predicate '$\exists overlap.\top$' of complexity 2, and the second as '$overlap$' of complexity 1. More complex groundings are obtained in IPC Grid. For example, the following groundings have all complexity 4: the nullary '$armempty$' predicate that holds when the robot holds no key, is grounded on the derived O2D predicate '$key \sqsubseteq \exists overlap.\top$' (i.e., all keys are in cells); the unary predicate '$somecell(\cdot)$' that holds for a key k if k is in some cell, is grounded on '$key \sqcap \exists overlap.\top$', and the binary predicate '$match(\cdot,\cdot)$' that holds when key k has the shape of the lock at cell c, is grounded on '$shape \circ shape^{-1}$' (a binary relation that holds for two objects of the same shape).

Planning with Learned Grounded Domains

The computational value of learning grounded domains $\langle D, \sigma \rangle$ can be illustrated by using them to solve new instances $P = \langle D, I \rangle$ expressed in terms of pairs of O2D states, s_0 and s_g, for the initial and goal situations encoded as $h(s_0)$ and $h(s_g)$ for $h = h^{\sigma}_{D}$. The new instances may involve sets of objects O that are much larger than those used in training. The resulting instances are standard and can be solved with any off-the-shelf planner.

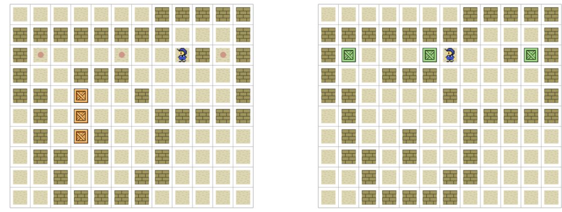


Figure 4: Depiction of initial and goal O2D states for a large Sokoban instance. Optimal plans of length 156 are found using the original "hidden" domain D_O and the learned grounded domain D_L.

A plan $\pi = \langle a_0, \ldots, a_n \rangle$ that solves such an instance P can be used to compute the corresponding sequence of O2D states $s_0, \ldots, s_{n+1}$, with $s_G = s_{n+1}$, as follows. If $\bar{s}_0, \ldots, \bar{s}_{n+1}$ are the planning states visited by π with $\bar{s}_0 = h(s_0)$, and the label of a_0 is α, s_1 is the possible α-successor of s_0 such that $h(s_1) = \bar{s}_1$. The successors $s_2, \ldots, s_{n+1}$ are selected in the same way.[5] This method of "applying" the plans obtained from the learned grounded domain provides an extra verification: if there is no α-successor s_{i+1} with $h(s_{i+1}) = \bar{s}_{i+1}$ or $s_{n+1} \neq s_g$, the learned domain or its grounding is not generalizing to the new instance. The fact that this does not happen in the experiments below is thus additional evidence that the learned grounded domains are correct.[6]

Figure 4 depicts the initial and goal O2D states s_0 and s_g of a large Sokoban instance. A plan π of length 288 was obtained from the planning instance $P = \langle D, I \rangle$, where D is the learned domain, and $h(s_0)$ and $h(s_g)$ replace s_0 and s_g. The plan was found with Pyperplan (Alkhazraji et al. 2020) running a greedy best-first search guided with the additive heuristic. For each state $\bar{s}_i$ generated by the plan π, a matching O2D state s_i was found as above, and $s_n = s_g$. The same verification was carried out in Blocks4ops instances with 7, 10, 15, 20 and 25 blocks, some producing plans with up to 121 actions.

We also compared the performance of planners on instances $P = \langle D_O, I \rangle$, where D_O is the hidden domain used to generate the data and $I = \langle O, \bar{s}_0, \{\bar{s}_g\} \rangle$, and the corresponding instances $P' = \langle D_L, I' \rangle$, where D_L is the learned domain and I' replaces the initial and goal states $\bar{s}$ by $h(g(\bar{s}))$, a mapping that uses the "rendering" function used to generate the O2D states (see appendix) and the learned function $h = h^{D}_{\sigma}$. If the learned domains are correct, the state graphs associated to P and P' should be isomorphic and the optimal plans should have the same length (but the plans themselves do not have to be the same). We tested this

[5] The method assumes a simulator that given an O2D state s produces the possible α-successors of s. In the experiments, the simulator is determined by a "hidden" domain and the function $g(\cdot)$ that maps planning states into O2D states, but a different one could be used potentially where the O2D states are obtained from images.

[6] Equivalence can also be proved formally.

in three large instances of Sokoban and of Blocks4ops using an optimal planner that runs A* with the LM-cut heuristic (Helmert and Domshlak 2009). For the Sokoban instance shown in Fig. 4, an optimal plan of length 156 was found in 27 seconds for P and in 54 seconds for P^0. For two other instances, optimal plans of length 134 and 135 were found in 65 and 849 seconds for P, and in 130 and 1,520 seconds for P^0. Similar results were obtained for the Blocks4ops instances.

Summary

We have introduced a formulation for learning crisp and meaningful first-order planning domains from parsed visual representations that are not far from those produced by object detection modules. For this, the formulation for learning domains (action schemas and predicates) from the structure of the state space (Bonet and Geffner 2020; Rodriguez et al. 2021) was taken to a new setting where the traces do not have to be complete and the states observed are not black boxes but parsed images in O2D. Two results are that the learned planning representations are grounded in O2D states, and hence new problems can be given in terms of pairs of O2D states representing the initial and goal situations, and that the learning scheme scales up better than previous ones, enabling us to learn more challenging domains like the Sliding-tile puzzle, IPC Grid, and Sokoban. We have also run planning experiments using the learned domains and their grounding functions that illustrate that the learned domains can be used with off-the-shelf planners and are not too different than the domains that are written and grounded by hand.

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Alkhazraji, Y.; Frorath, M.; Grützner, M.; Helmert, M.; Liebetraut, T.; Mattmüller, R.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; and Wülfing, J. 2020. Pyperplan. https://doi.org/10.5281/zenodo.3700819.

Asai, M. 2019. Unsupervised Grounding of Plannable First-Order Logic Representation from Images. In *Proc. ICAPS*.

Asai, M.; and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *AAAI*.

Asai, M.; and Muise, C. 2020. Learning Neural-Symbolic Descriptive Planning Models via Cube-Space Priors: The Voyage Home (to STRIPS). In *Proc. IJCAI*.

Baader, F.; Horrocks, I.; and Sattler, U. 2008. *Handbook of Knowledge Representation*, chapter Description Logics. Elsevier.

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

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

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

Cohn, A. G.; and Renz, J. 2008. Qualitative spatial representation and reasoning. *Foundations of Artificial Intelligence*, 3: 551–596.

Cresswell, S. N.; and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *Twenty-First International Conference on Automated Planning and Scheduling*.

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.

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.; and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge U.P.

Gregory, P.; and Lindsay, A. 2016. Domain model acquisition in domains with action costs. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*.

Harnad, S. 1990. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3): 335–346.

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

Helmert, M.; and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway. In *Proc. International Conference on Automated Planning and Scheduling*, volume 9, 162–169.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61: 215–289.

Lamanna, L.; Saetti, A.; Serafini, L.; Gerevini, A.; and Traverso, P. 2021. Online Learning of Action Models for PDDL Planning. In *Proc. IJCAI*, 19–27.

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

Lindsay, A. 2021. Reuniting the LOCM Family: An Alternative Method for Identifying Static Relationships. In *ICAPS 2021 KEPS Workshop*.

Locatello, F.; Weissenborn, D.; Unterthiner, T.; Mahendran, A.; Heigold, G.; Uszkoreit, J.; Dosovitskiy, A.; and Kipf, T. 2020. Object-centric learning with slot attention. *NeurIPS*.

Mourao, K.; Zettlemoyer, L.; Petrick, R.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Proc. UAI*, 614–623.

Redmon, J.; Divvala, S.; Girshick, R.; and Farhadi, A. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 779–788.

Redmon, J.; and Farhadi, A. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 7263–7271.

Rodriguez, I. D.; Bonet, B.; Romero, J.; and Geffner, H. 2021. Learning First-Order Representations for Planning from Black-Box States: New Results. In *KR*. ArXiv preprint arXiv:2105.10830.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym environments from PDDL problems. *arXiv preprint arXiv:2002.06432*.

Stern, R.; and Juba, B. 2017. Efficient, safe, and probably approximately complete learning of action models. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 4405–4411.

Verma, P.; Marpally, S. R.; and Srivastava, S. 2021. Asking the Right Questions: Learning Interpretable Action Models Through Query Answering. In *Proc. AAAI*, 12024–12033.

Walsh, T. J.; and Littman, M. L. 2008. Efficient learning of action schemas and web-service descriptions. In *AAAI*, volume 8, 714–719.

Xie, Y.; Li, M.; Yu, S.; and Littman, M. L. 2022. Online learning in non-cooperative configurable Markov decision process. In *AAAI-22 Workshop on Reinforcement Learning in Games*.

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.

Zhuo, H. H.; and Kambhampati, S. 2013. Action-model acquisition from noisy plan traces. In *Proc. IJCAI*.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18): 1540–1569.

Appendix

This appendix contains the proofs of the theorems, further details about the data generation, a full description of the grounded domains learned, the full code of the learner (ASP program $T_\beta(\mathcal{D},\mathcal{P})$), and additional details of the verifier and of the function $g(\cdot)$ used to generate the data from hidden planning domains.

Proofs

Let us recall the definition and theorem statements:

Definition 2 (Learning Task)**.** *Let $\mathcal{D} = \langle \mathcal{T}, \mathcal{S}, A, F\rangle$ be the input data, and let $\mathcal{P}$ be a pool of grounded predicates. The* ***learning task*** *$L(\mathcal{D},\mathcal{P})$ is to obtain a (simplest) grounded domain $\langle D,\sigma\rangle$ with one action schema per label α in A such that the resulting abstraction function $h = h^D_\sigma$ complies with the following two constraints:*

C1. If $s \neq s'$, then $h(s) \neq h(s')$, for $s, s' \in \mathcal{T}$; and
C2. $F^D_\alpha(h(s)) = \{\{h(s') \mid s' \in F_\alpha(s)\}\}$ for $s \in \mathcal{T}$, $\alpha \in A$.

For a given dataset $\mathcal{D} = \langle \mathcal{T}, \mathcal{S}, L, F\rangle$, the data graph $G_{\mathcal{D}}$ has vertex set $V_{\mathcal{D}} = \mathcal{S}$ and labeled edges $E_{\mathcal{D}} = \{(s,\alpha,s') \mid s \in \mathcal{S}, s' \in F_\alpha(s), \alpha \in L\}$. On the other hand, for planning domain D and function h that maps states s in $\mathcal{S}$ into planning states $\bar{s}$ in D, the planning graph has as vertex set V_h the set of reachable planning states from $\{h(s) \mid s \in \mathcal{S}\}$, and as labeled edges E_h the set $\{(\bar{s},\alpha,\bar{s}') \mid \bar{s} \in V_h, a \in A(\bar{s}), label(a) = \alpha, \bar{s}' \in F^D_\alpha(\bar{s})\}$.

Theorem 3. *If $\langle D,\sigma\rangle$ is a solution of the learning task $L(\mathcal{D},\mathcal{P})$ and $\mathcal{T} = \mathcal{S}$, the data and planning graphs $G_{\mathcal{D}}$ and G_h for $h = h^D_\sigma$ are isomorphic.*

Proof. We first show that the function h is a bijection from $V_{\mathcal{D}}$ onto V_h, and then show that the multisets of labeled edges are preserved by h.

By construction of D, the set of labels in both graphs are equal, and by constraint C1 in Def. 2, the function $h : V_{\mathcal{D}} \to V_h$ is 1-1. To show that h is onto, we show $|V_h| \leq |V_{\mathcal{D}}|$. For a proof by contradiction, suppose $|V_{\mathcal{D}}| < |V_h|$. Then, either V_h contains a vertex not reachable from $\{h(s) \mid s \in \mathcal{S}\}$, or there is a vertex $\bar{s}$ in V_h and label α in L such that

$$|\{\{h(s') \mid s' \in F_\alpha(s)\}\}| < |F^D_\alpha(h(s))|\,.$$

The first case is impossible by definition of G_h. In the second case, since $s' \in F_\alpha(s)$ implies $h(s') \in F^D_\alpha(h(s))$ (by constraint C2), then there is a state $s' \in \mathcal{S}$ such that $s' \notin F_\alpha(s)$ and $h(s') \in F^D_\alpha(h(s))$, which also contradicts C2.

Finally, to show that h preserves edges, let s and s' be two states in $\mathcal{S}$, and let α be an action label. If $(s,\alpha,s') \in G_{\mathcal{D}}$, then $h(s') \in F^D_\alpha(h(s))$ by $C2$. Likewise, if $(h(s),\alpha,h(s')) \in G_h$, then $s' \in F_\alpha(s)$ also by $C2$. Hence, h preserves edges and $G_{\mathcal{D}}$ and G_h are isomorphic. □

Theorem 4. *Let D be a (hidden) planning domain, let $\mathcal{D} = \langle \mathcal{T}, \mathcal{S}, L, F\rangle$ be a dataset, and let g be a 1-1 function that maps planning states $\bar{s}$ in D into O2D states $g(\bar{s})$ such that $F_\alpha(g(\bar{s})) = \{\{g(\bar{s}') \mid \bar{s}' \in F^D_\alpha(\bar{s})\}\}$ for $g(\bar{s}) \in \mathcal{T}$ and α in L. If there is a grounding function σ for the predicates in D over a pool $\mathcal{P}$ such that $h = h^D_\sigma$ is the right inverse of g on $\mathcal{T}$ (i.e., $g(h(s)) = s$ for $s \in \mathcal{T}$), then $\langle D,\sigma\rangle$ is a solution for the learning task $L(\mathcal{D},\mathcal{P})$.*

Proof. We need to show that the grounded domain $\langle D,\sigma\rangle$ complies with the constraints C1 and C2 in Definition 2.

For C1, let s and s' be different states in $\mathcal{T}$. If $h(s) = h(s')$, then $s = g(h(s)) = g(h(s')) = s'$ by the condition on g.

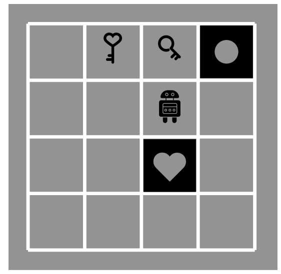
Let $s \in \mathcal{T}$ be an O2D state, and let $\alpha \in L$ be an action label. First notice that for planning state $\bar{s}$, $g(h(g(\bar{s}))) = g(\bar{s})$ implies $h(g(\bar{s})) = \bar{s}$ and thus h is a left inverse of g. Then,

$$\begin{aligned}
F^D_\alpha(h(s)) &= \{\{\bar{s}' \mid \bar{s}' \in F^D_\alpha(h(s))\}\} && \text{(definition)}\\
&= \{\{\bar{s}' \mid g(\bar{s}') \in F_\alpha(g(h(s)))\}\} && \text{(def. } F_\alpha \text{ in Thm)}\\
&= \{\{\bar{s}' \mid g(\bar{s}') \in F_\alpha(s)\}\} && \text{(right inv.)}\\
&= \{\{h(g(\bar{s}')) \mid g(\bar{s}') \in F_\alpha(s)\}\} && \text{(left inv.)}\\
&= \{\{h(s') \mid s' \in F_\alpha(s)\}\}\,. && \text{(def. } F_\alpha \text{ in Thm)}
\end{aligned}$$

Therefore, constraint C2 is satisfied as well. □


```
% object and types
tile(t1). tile(t2).
cell(c1_1). cell(c1_2).
...
% relations
overlap(t1,c1_1).
below(c2_1,c_1_1).
...
% shapes
shape(d1,rectangle).
...
```

Figure 5: Slidingtile scene and corresponding O2D state.



```
% objects and types
robot(r). key(k0). key(k1).
cell(c1_1).
blackcell(c1_4).
...
% relations
overlap(k0,c1_2).
overlap(r,c2_3).
below(c3_1,c_2_1).
...
% shapes
shape(k0,heart).
shape(c1_4,circle).
shape(k1,circle).
shape(c3_3,heart).
...
```

Figure 6: Grid scene and corresponding O2D state.

Data Generation: Details

Blocks3ops and Blocks4ops. Two encodings of the classical planning domain, where stackable blocks need to be reassembled on a table by a robot. Instances are parametrized by the number n of blocks. The instances in the dataset have $n = 1, \ldots, 5$ blocks. Blocks3ops has 3 action labels (Stack, Newtower, and Move) while Blocks4ops has 4 (Pickup, Putdown, Unstack, and Stack). O2D states are defined based on the corresponding PDDLGym state images for this domain, as illustrated in Figure 2.

Hanoi1op and Hanoi4ops. Two encodinfs of the Tower of Hanoi problem with arbitraty number of pegs and disks. The datasets in both cases contain the instances for 3 pegs and n disks, $n = 1, \ldots, 5$.

Hanoi1op involve a single action label Move while Hanoi4ops has 4 labels: MoveFromPegToPeg, MoveFromPegToDisk, MoveFromDiskToPeg, and MoveFromDiskToDisk.

Sliding Tile. The generalization of the 15-puzzle problem over rectangular grids of arbitrary dimensions, parametrized as $r \quad c$ where r and c are the number of rows and columns.

The dataset contains instances $r \quad c$ such that the number of cells $rc \quad 6$. The action labels are MoveUp, MoveRight, MoveDown, and MoveLeft. O2D states are defined based on images such as the one illustrated in Figure 5.

IPC Grid In this planning problem from the Int. Planning Competition (IPC), there is a robot that moves within a rectangular grid where cells may be locked, but that can be opened with matching keys, where a key and a cell match if they have the same shape. Keys can be picked and dropped by the robot, and locked cells can be opened with the right key from an adjacent cell. The goal is to have some of the keys at specified locations. Instances are parametrized by the number of rows r and columns c of the grid, the number of key/cell shapes s, the number of keys k, and the number of locked cells ℓ. The instances used for learning are generated with $r \quad 2$, $c \quad 2$, $s \quad 2$, $k \quad 2$ and $\ell \quad 1$. For each combination of parameters, one instance is generated, in which the locations of objects is randomized. The action space has 10 labels: MoveUp, MoveRight, MoveDown, MoveLeft, Pickup, Putdown, UnlockFromAbove, UnlockFromRight, UnlockFromBelow, and UnlockFromLeft. O2D states are defined based on images such as the one illustrated in Figure 6.

Sokoban1 and Sokoban2. A puzzle where a player (Sokoban) pushes boxes (crates) around in a warehouse (represented as a grid), trying to get them to designated storage locations. Instances are parametrized as (r, c, b) for the number of rows r, the number of columns c, and the number of boxes b located on the grid, yet the parameter does not determine the instance because the crates and Sokoban may be in different locations initally and at the goal. Two different datasets are considered, one with many but smaller instances, and the other with fewer but bigger instances. The dataset for Sokoban1 consits of 94 instances for (r, c, b) in $f(1,5,b),(2,3,b),(3,2,b),(5,1,b)g$ for $b = 0, 1, 2$, and one extra (larger) instance for $(4, 5, 2)$. The dataset for Sokoban2 consits of 24 instances for (r, c, b) in $f(r,5,b)j\,r \quad 2 \quad f1,2,4,5gg \; [\; f(5,c,b)j\,c \quad 2 \quad f1,\ldots,5gg$ with $b = 0, 1, 2$. The action space for Sokoban has 8 labels: MoveUp, MoveRight, MoveDown, MoveLeft, PushUp, PushRight, PushDown, and PushLeft.

Mapping STRIPS States into O2D States

For each planning instance in the data pool expressed as a pair of domain and instance PDDL files, the full reachable state space is enumerated. Then, for each reachable state $\bar{s}$, a "rendering" function $g(\;)$ that maps STRIPS to O2D states is applied. The rendering function is specified by a set of DATALOG rules that say how the visual elements of the O2D scene are obtained.

The rules used, in JSON format, are shown in Figure 7. They contain all the information about how planning states are mapped into scene representations in O2D.

Learned Grounded Domains

Figures 8–14 show the learned grounded domains for all the learning tasks in the experiments. In each case, we show the final and optimal value of the (optimized) cost function (lines 357–367 in Fig. 18), the grounded predicates from the pool P that make up the learned domain, the action schemas in the domain, the constants (if any), and some stats about the incremental solver.

In these models, grounded predicates P are directly represented by their grounding $\sigma(P)$ as computed by the learner, and their description is given as `<predicate>/<arity>` in the top part of each domain. The notation to represent concepts, roles and predicates is as follows, where $d(\cdot)$ is the denotation function:

– O2D concept C and role R denoted by C and R, resp.,
– Concept '$\top$' denoted by `Top`,
– Concept '$\exists R.C$' denoted by `ER[`$d(R)$`,`$d(C)$`]`,
– Concept '$C \sqcap C'$' denoted by `INTER[`$d(C)$`,`$d(C')$`]`,
– Role 'R^{-1}' denoted by `INV[`$d(R)$`]`,
– Role '$R \circ R'$' denoted by `COMP[`$d(R)$`,`$d(R')$`]`, and
– Predicate '$C \sqsubseteq C'$' denoted by `SUBSET[`$d(C)$`,`$d(C')$`]`.

Recall that concepts and roles directly yield predicates of arity 1 and 2 respectively, while nullary predicates are obtained with the subset construction (i.e., $C \sqsubseteq C'$).

Implementation Details

Full ASP Program The code in ASP for learning the instances $P_i = \langle D, I_i \rangle$ from multiple input graphs G_i, using a pool of predicates $\mathcal{P}$, is shown in Figures 15–18. Each graph G_i is assumed to be encoded using the atoms `node(I,S)` and `tlabel(I,T,L)` where `S` and `T=(S1,S2)` denote nodes and transitions in the graph G_i with index `I`, and `L` denotes the corresponding action label.

At each step of incremental learning, each instance `I` and node `S` from this instance that must be taken into account at that step is marked with an atom `relevant(I,S)`. Truth values `V` of ground atoms `(P,OO)` from $\mathcal{P}$, in the state `S` of instance `I`, are encoded in the input with atoms `val(I,(P,OO),S,V)`. When computing the truth values for such atoms, *redundant* predicates from $\mathcal{P}$ are pruned. A predicate is redundant when its denotation over all states in the dataset is the same as some previously considered predicate; i.e., given some enumeration of the predicates in the pool, the predicate p_i is redundant iff there is p_j such that for each state s of each instance, $p_i^s = p_j^s$, $j < i$. Moreover, for each instance `I`, we compute the predicates that are static over that instance, mark them with the fact `f_static(I,P)`, and encode their truth value `V` in all states of that instance with a single fact `val(I,(P,OO2),V)`. Each predicate `P` in $\mathcal{P}$ is marked with the fact `feature(P)`, and their arity `N` is encoded by fact `f_arity(P,N)`. The number of action schemas is set to the number of action labels and the objects are extracted from the valuation of the concept $\top$ (line 29). The max number of chosen predicates is set to the value of the constant `num_predicates` (12 by default), while the max arity of actions is set to the value of the constant `max_arity` (3 by default, but with max value of 4 for the code shown).

Exploiting the fact that the predicates in the pool have a maximum arity of 2, the applicability relation for grounded actions as well as the successor function are factored (lines 147–249). This makes the code longer but results in improved grounding and solving times.

Verifier The verifier, written in Python, receives the input data graph $G_{\mathcal{D}}$ for the instance together with the learned grounded model D, and outputs a subset Δ of states in $G_{\mathcal{D}}$: either $\Delta = \emptyset$ meaning *successful verification*, $\Delta = \{s, s'\}$ of two such states that are identical modulo the grounded predicates in the model (cf. constraint C1 in 2), or a non-empty subset of states in $G_{\mathcal{D}}$ for which constraint C2 does not hold. The verifier is called from the incremental solver that then uses Δ to extend the learning dataset $\mathcal{D}$ as described in the paper.

The verifier works as follows. First, C1 is checked for all pair of states in $G_{\mathcal{D}}$. If for some pair (s, s') C1 fails, the verifier terminates and outputs $\{s, s'\}$. Otherwise, for each state s in $G_{\mathcal{D}}$, the verifier checks that each transition (s, s') with label α in $G_{\mathcal{D}}$ has a matching transition $(h(s), h(s'))$ in the learned model D via a grounded action with label α, and vice versa, that each transition $(h(s), \bar{s}')$ in D via a grounded action with label α has a matching transition (s, s'') in $G_{\mathcal{D}}$ with label α such that $h(s'') = \bar{s}'$; if some of these two checks fails, the state s is added to the output set for the verifier.

Appendix continues with figures in the next few pages.

```
f "blocks3ops" :
    f "constants" : ["rectangle", "t"],
      "facts"     : [ ["table",["t"]] ],
      "rules"     : f "block"      : [ ["block(X)",           ["ontable(X)"]],
                                       ["block(X)",           ["on(X,Y)"]] ],
                      "below"      : [ ["below(X,Y)",         ["on(Y,X)"]],
                                       ["below(X,Y)",         ["ontable(Y)", "table(X)"]] ],
                      "smaller"    : [ ["smaller(X,Y)",       ["block(X)", "table(Y)"]],
                                       ["smaller(X,Z)",       ["smaller(X,Y)", "smaller(Y,Z)"]] ],
                      "shape"      : [ ["shape(X,rectangle)", ["object(X)"]] ]
                    g g,
   "blocks4ops" :
    f "constants" : ["rectangle", "r", "t"],
      "facts"     : [ ["robot",["r"]], ["table",["t"]] ],
      "rules"     : f "block"      : [ ["block(X)",           ["ontable(X)"]],
                                       ["block(X)",           ["on(X,Y)"]] ],
                      "overlap"    : [ ["overlap(X,Y)",       ["holding(X)", "robot(Y)"]],
                                       ["overlap(Y,X)",       ["overlap(X,Y)"]] ],
                      "below"      : [ ["below(X,Y)",         ["on(Y,X)"]],
                                       ["below(X,Y)",         ["ontable(Y)", "table(X)"]] ],
                      "smaller"    : [ ["smaller(X,Y)",       ["block(X)", "table(Y)"]],
                                       ["smaller(X,Y)",       ["block(X)", "robot(Y)"]],
                                       ["smaller(X,Y)",       ["robot(X)", "table(Y)"]],
                                       ["smaller(X,Z)",       ["smaller(X,Y)", "smaller(Y,Z)"]] ],
                      "shape"      : [ ["shape(X,rectangle)", ["object(X)"]] ]
                    g g,
  "hanoi1op" :
    f "constants" : ["rectangle"],
      "facts"     : [],
      "rules"     : f "overlap"    : [ ["overlap(X,Y)",       ["disk(X)", "peg(Y)", "on(X,Y)"]],
                                       ["overlap(Y,X)",       ["overlap(X,Y)"]] ],
                      "below"      : [ ["below(X,Y)",         ["on(Y,X)", "disk(X)", "disk(Y)"]],
                                       ["below(X,Y)",         ["on(Y,X)", "peg(X)", "disk(Y)"]] ],
                      "shape"      : [ ["shape(X,rectangle)", ["object(X)"]] ]
                    g g,
  "hanoi4ops" : f "defer-to" : "hanoi1op" g,
  "slidingtile" :
    f "constants" : ["rectangle"],
      "facts"     : [],
      "rules"     : f "cell"       : [ ["cell(X)",            ["position(X)"]] ],
                      "overlap"    : [ ["overlap(X,Y)",       ["at(X,Y)"]],
                                       ["overlap(Y,X)",       ["overlap(X,Y)"]] ],
                      "shape"      : [ ["shape(X,rectangle)", ["object(X)"]] ]
                    g g,
  "grid" :
    f "constants" : ["r"],
      "facts"     : [ ["robot",["r"]] ],
      "rules"     : f "cell"       : [ ["cell(X)",            ["place(X)", "open(X)"]] ],
                      "blackcell"  : [ ["blackcell(X)",       ["locked(X)"]] ],
                      "overlap"    : [ ["overlap(X,r)",       ["at_robot(X)"]],
                                       ["overlap(X,Y)",       ["at(X,Y)"]],
                                       ["overlap(Y,X)",       ["overlap(X,Y)"]] ],
                      "smaller"    : [ ["smaller(X,Y)",       ["robot(X)", "place(Y)"]],
                                       ["smaller(X,Y)",       ["key(X)", "place(Y)"]],
                                       ["smaller(X,Z)",       ["smaller(X,Y)", "smaller(Y,Z)"]] ],
                      "shape"      : [ ["shape(X,S)",         ["lock_shape(X,S)"]],
                                       ["shape(X,S)",         ["key_shape(X,S)"]] ]
                    g g,
  "sokoban1" :
    f "constants" : ["sokoban1", "rectangle", "sokoshape"],
      "facts"     : [ ["sokoban",["sokoban1"]] ],
      "rules"     : f "cell"       : [ ["cell(X)",            ["leftof(X,Y)"]],
                                       ["cell(Y)",            ["leftof(X,Y)"]],
                                       ["cell(X)",            ["below(X,Y)"]],
                                       ["cell(Y)",            ["below(X,Y)"]],
                                       ["cell(Y)",            ["at(X,Y)"]] ],
                      "overlap"    : [ ["overlap(X,Y)",       ["at(X,Y)"]],
                                       ["overlap(Y,X)",       ["overlap(X,Y)"]] ],
                      "left"       : [ ["left(X,Y)",          ["leftof(X,Y)"]] ],
                      "shape"      : [ ["shape(X,sokoshape)", ["sokoban(X)"]],
                                       ["shape(X,rectangle)", ["cell(X)"]],
                                       ["shape(X,rectangle)", ["crate(X)"]] ]
                    g g,
  "sokoban2" : f "defer-to" : "sokoban1" g
g
```

Figure 7: JSON specification of the O2D rendering function $g(\,)$ that is used to generate the datasets from the STRIPS instances. For each reachable STRIPS state $\bar{s}$ in such an instance, the "rules" are applied until reaching a fixpoint, where a rules consists of a head (single O2D atom) and a body (list of atoms). The resulting O2D state is obtained by preserving the resulting O2D atoms, and removing all the STRIPS atoms except the static ones that specify types; i.e., `robot`, `block`, `table`, `sokoban`, `crate`, `key`, and `tile`.

```
Optimization: (10,5,0,10,8)
1 constant(s): t
2 predicate(s): ER[below,Top]/1, INV[below]/2

Stack(1,2):
  pre: :ER[below,Top](1), INV[below](1,t), :ER[below,Top](2)
  eff: ER[below,Top](2), :INV[below](1,t), INV[below](1,2)
Newtower(1,2):
  pre: :ER[below,Top](1), INV[below](1,2)
  eff: :ER[below,Top](2), INV[below](1,t), :INV[below](1,2)
Move(1,2,3):
  pre: :ER[below,Top](1), INV[below](1,2), :ER[below,Top](3)
  eff: :ER[below,Top](2), ER[below,Top](3), :INV[below](1,2), INV[below](1,3)

#calls=5, solve_wall_time=1.97, solve_ground_time=1.92, verify_time=0.84, elapsed_time=2.97
```

Figure 8: Learned grounded domain for Blocks3ops

```
Optimization: (10,7,0,14,9)
2 constant(s): r, t
3 predicate(s): ER[overlap,Top]/1, INTER[ER[below,Top],block]/1, below/2

Pickup(1):
  pre: :ER[overlap,Top](r), :INTER[ER[below,Top],block](1), below(t,1)
  eff: ER[overlap,Top](r), ER[overlap,Top](1), :below(t,1)
Putdown(1):
  pre: ER[overlap,Top](1)
  eff: :ER[overlap,Top](r), :ER[overlap,Top](1), below(t,1)
Unstack(1,2):
  pre: :ER[overlap,Top](r), :INTER[ER[below,Top],block](1), below(2,1)
  eff: ER[overlap,Top](r), ER[overlap,Top](1), :INTER[ER[below,Top],block](2), :below(2,1)
Stack(1,2):
  pre: ER[overlap,Top](1), :INTER[ER[below,Top],block](2)
  eff: :ER[overlap,Top](r), :ER[overlap,Top](1), INTER[ER[below,Top],block](2), below(2,1)

#calls=7, solve_wall_time=23.66, solve_ground_time=23.37, verify_time=29.42, elapsed_time=53.70
```

Figure 9: Learned grounded domain for Blocks4ops

```
Optimization: (4,5,3,4,4)
3 predicate(s): ER[below,Top]/1, below/2, smaller/2
1 static predicate(s): smaller/2

Move(1,2,3):
  static: smaller(1,2)
  pre: :ER[below,Top](1), :ER[below,Top](2), below(3,1)
  eff: ER[below,Top](2), :ER[below,Top](3), below(2,1), :below(3,1)

#calls=4, solve_wall_time=1.59, solve_ground_time=1.53, verify_time=0.44, elapsed_time=2.16
```

Figure 10: Learned grounded domain for Hanoi1op

```
Optimization: (16,5,5,16,22)
4 predicate(s): ER[below,Top]/1, ER[smaller,Top]/1, INV[below]/2, INV[smaller]/2
2 static predicate(s): ER[smaller,Top]/1, INV[smaller]/2

MoveFromPegToPeg(1,2,3):
  static: :ER[smaller,Top](1), :INV[smaller](1,3)
  pre: :ER[below,Top](2), INV[below](2,1), :ER[below,Top](3)
  eff: :ER[below,Top](1), ER[below,Top](3), :INV[below](2,1), INV[below](2,3)
MoveFromPegToDisk(1,2,3):
  static: :INV[smaller](1,2), INV[smaller](3,2), :ER[smaller,Top](3)
  pre: :ER[below,Top](1), :ER[below,Top](2), INV[below](1,3)
  eff: ER[below,Top](2), :ER[below,Top](3), INV[below](1,2), :INV[below](1,3)
MoveFromDiskToPeg(1,2,3):
  static: ER[smaller,Top](2), :ER[smaller,Top](3)
  pre: :ER[below,Top](1), INV[below](1,2), :ER[below,Top](3)
  eff: :ER[below,Top](2), ER[below,Top](3), :INV[below](1,2), INV[below](1,3)
MoveFromDiskToDisk(1,2,3):
  static: :INV[smaller](1,2), ER[smaller,Top](2), ER[smaller,Top](3)
  pre: :ER[below,Top](1), :ER[below,Top](2), INV[below](1,3)
  eff: ER[below,Top](2), :ER[below,Top](3), INV[below](1,2), :INV[below](1,3)

#calls=6, solve_wall_time=14.23, solve_ground_time=12.67, verify_time=0.59, elapsed_time=15.06
```

Figure 11: Learned grounded domain for Hanoi4ops

```
Optimization: (16,5,6,24,12)
4 predicate(s): ER[overlap,Top]/1, overlap/2, INV[left]/2, INV[below]/2
2 static predicate(s): INV[left]/2, INV[below]/2

MoveUp(1,2,3):
  static: INV[below](3,2)
  pre: overlap(1,2), :ER[overlap,Top](3)
  eff: :ER[overlap,Top](2), ER[overlap,Top](3), :overlap(1,2), overlap(1,3), :overlap(2,1), overlap(3,1)
MoveRight(1,2,3):
  static: INV[left](3,2)
  pre: overlap(1,2), :ER[overlap,Top](3)
  eff: :ER[overlap,Top](2), ER[overlap,Top](3), :overlap(1,2), overlap(1,3), :overlap(2,1), overlap(3,1)
MoveDown(1,2,3):
  static: INV[below](2,3)
  pre: overlap(2,1), :ER[overlap,Top](3)
  eff: :ER[overlap,Top](2), ER[overlap,Top](3), :overlap(1,2), overlap(1,3), :overlap(2,1), overlap(3,1)
MoveLeft(1,2,3):
  static: INV[left](2,3)
  pre: overlap(2,1), :ER[overlap,Top](3)
  eff: :ER[overlap,Top](2), ER[overlap,Top](3), :overlap(1,2), overlap(1,3), :overlap(2,1), overlap(3,1)

#calls=6, solve_wall_time=3.00, solve_ground_time=2.89, verify_time=1.20, elapsed_time=4.43
```

Figure 12: Learned grounded domain for Sliding Tile

```
Optimization: (34,8,11,28,42)
1 constant(s): r
8 predicate(s): SUBSET[key,ER[overlap,Top]]/0, cell/1, ER[smaller,Top]/1, INTER[key,ER[overlap,Top]]/1,
                below/2, overlap/2, INV[left]/2, COMP[shape,INV[shape]]/2
4 static predicate(s): ER[smaller,Top]/1, below/2, INV[left], COMP[shape,INV[shape]]/2

MoveUp(1,2):
  static: below(1,2)
  pre: overlap(r,1), cell(2)
  eff: :overlap(r,1), overlap(r,2), :overlap(1,r), overlap(2,r)
MoveRight(1,2):
  static: INV[left](2,1)
  pre: overlap(1,r), cell(2)
  eff: :overlap(r,1), overlap(r,2), :overlap(1,r), overlap(2,r)
MoveDown(1,2):
  static: below(2,1)
  pre: overlap(1,r), cell(2)
  eff: :overlap(r,1), overlap(r,2), :overlap(1,r), overlap(2,r)
MoveLeft(1,2):
  static: INV[left](1,2)
  pre: overlap(r,1), cell(2)
  eff: :overlap(r,1), overlap(r,2), :overlap(1,r), overlap(2,r)
Pickup(1,2):
  pre: SUBSET[key,ER[overlap,Top]], overlap(1,r), overlap(1,2)
  eff: :SUBSET[key,ER[overlap,Top]], :INTER[key,ER[overlap,Top]](2), :overlap(1,2), :overlap(2,1)
Putdown(1,2):
  static: ER[smaller,Top](2)
  pre: overlap(1,r), :INTER[key,ER[overlap,Top]](2)
  eff: SUBSET[key,ER[overlap,Top]], INTER[key,ER[overlap,Top]](2), overlap(1,2), overlap(2,1)
UnlockFromAbove(1,2,3):
  static: below(2,1), COMP[shape,INV[shape]](2,3)
  pre: :SUBSET[key,ER[overlap,Top]], overlap(1,r), :cell(2), :INTER[key,ER[overlap,Top]](3)
  eff: cell(2)
UnlockFromRight(1,2,3):
  static: INV[left](2,1), COMP[shape,INV[shape]](1,3), ER[smaller,Top](3)
  pre: :cell(1), overlap(r,2), :INTER[key,ER[overlap,Top]](3)
  eff: cell(1)
UnlockFromBelow(1,2,3):
  static: below(2,1), COMP[shape,INV[shape]](1,3)
  pre: :SUBSET[key,ER[overlap,Top]], :cell(1), overlap(2,r), :INTER[key,ER[overlap,Top]](3)
  eff: cell(1)
UnlockFromLeft(1,2,3):
  static: INV[left](1,2), COMP[shape,INV[shape]](3,1), ER[smaller,Top](3)
  pre: :cell(1), overlap(r,2), :INTER[key,ER[overlap,Top]](3)
  eff: cell(1)

#calls=27, solve_wall_time=4229.67, solve_ground_time=3536.23, verify_time=2404.87, elapsed_time=6653.03
```

Figure 13: Learned grounded domain for IPC Grid

```
Optimization: (32,5,6,64,32)
1 constant(s): sokoban1
4 predicate(s): ER[overlap,Top]/1, below/2, overlap/2, INV[left]/2
2 static predicate(s): below/2, INV[left]/2

MoveUp(1,2):
  static: below(1,2)
  pre: overlap(1,sokoban1), :ER[overlap,Top](2)
  eff: :ER[overlap,Top](1), ER[overlap,Top](2), :overlap(sokoban1,1), overlap(sokoban1,2), :overlap(1,sokoban1),
       overlap(2,sokoban1)
MoveRight(1,2):
  static: INV[left](2,1)
  pre: overlap(1,sokoban1), :ER[overlap,Top](2)
  eff: :ER[overlap,Top](1), ER[overlap,Top](2), :overlap(sokoban1,1), overlap(sokoban1,2), :overlap(1,sokoban1),
       overlap(2,sokoban1)
MoveDown(1,2):
  static: below(2,1)
  pre: overlap(sokoban1,1), :ER[overlap,Top](2)
  eff: :ER[overlap,Top](1), ER[overlap,Top](2), :overlap(sokoban1,1), overlap(sokoban1,2), :overlap(1,sokoban1),
       overlap(2,sokoban1)
MoveLeft(1,2):
  static: INV[left](1,2)
  pre: overlap(1,sokoban1), :ER[overlap,Top](2)
  eff: :ER[overlap,Top](1), ER[overlap,Top](2), :overlap(sokoban1,1), overlap(sokoban1,2), :overlap(1,sokoban1),
       overlap(2,sokoban1)
PushUp(1,2,3,4):
  static: below(3,1), below(1,4)
  pre: overlap(1,2), overlap(3,sokoban1), :ER[overlap,Top](4)
  eff: :ER[overlap,Top](3), ER[overlap,Top](4), overlap(sokoban1,1), :overlap(sokoban1,3),
       overlap(1,sokoban1), :overlap(3,sokoban1), :overlap(1,2), :overlap(2,1), overlap(2,4), overlap(4,2)
PushRight(1,2,3,4):
  static: INV[left](1,3), INV[left](4,1)
  pre: overlap(2,1), overlap(sokoban1,3), :ER[overlap,Top](4)
  eff: :ER[overlap,Top](3), ER[overlap,Top](4), overlap(sokoban1,1), :overlap(sokoban1,3),
       overlap(1,sokoban1), :overlap(3,sokoban1), :overlap(1,2), :overlap(2,1), overlap(2,4), overlap(4,2)
PushDown(1,2,3,4):
  static: below(1,3), below(4,1)
  pre: overlap(2,1), overlap(sokoban1,3), :ER[overlap,Top](4)
  eff: :ER[overlap,Top](3), ER[overlap,Top](4), overlap(sokoban1,1), :overlap(sokoban1,3),
       overlap(1,sokoban1), :overlap(3,sokoban1), :overlap(1,2), :overlap(2,1), overlap(2,4), overlap(4,2)
PushLeft(1,2,3,4):
  static: INV[left](3,1), INV[left](1,4)
  pre: overlap(2,1), overlap(3,sokoban1), :ER[overlap,Top](4)
  eff: :ER[overlap,Top](3), ER[overlap,Top](4), overlap(sokoban1,1), :overlap(sokoban1,3),
       overlap(1,sokoban1), :overlap(3,sokoban1), : :overlap(1,2), overlap(2,1), overlap(2,4), overlap(4,2)

#calls=11, solve_wall_time=12565.02, solve_ground_time=5314.35, verify_time=165.19, elapsed_time=12740.43
```

Figure 14: Learned grounded domain for Sokoban

```
% Suggested call
% clingo -t 6 --sat-prepro=2 --time-limit=7200 <this-solver> <graph-files>

% Constants and options
#const num_predicates = 12.
#const max_action_arity = 3.
#const null_arg = (null,).
#const opt_equal_objects = 0.          % Allow same obj as argument for grounded actions
#const opt_allow_negative_precs = 1.   % Allow for negative preconditions
#const opt_fill = 1.                   % Fill in missing negative valuations for primitive predicates
#const opt_symmetries = 1.             % Some (simple) symmetry breaking

% Input Instances defined by instance/1, and graphs by tlabel/3 and node/2
% O2D features defined by feature/1, f_arity/2, f_static/2, fval/3, and fval/4
nullary(F)   :- feature(F), f_arity(F,1), 1 f fval(I,(F,null_arg),0..1) g.                % Explicit zero_arity predicates
nullary(F)   :- feature(F), f_arity(F,1), 1 f fval(I,(F,null_arg),S,0..1) : node(I,S) g.  % Explicit zero_arity predicates
p_arity(F,N) :- feature(F), f_arity(F,N), not nullary(F).                                 % Explicit zero_arity predicates
p_arity(F,0) :- nullary(F).                                                                % Explicit zero_arity predicates
:- p_arity(F,N), nullary(F), N > 0.                                                        % Explicit zero_arity predicates

% Relevant nodes (all relevant by default; overriden by incremental solver)
#defined filename/1.
#defined partial/2.
relevant(I,S) :- node(I,S), not partial(I,File) : filename(File).

% Actions and objects (objects come from denotation of concept Top)
action(A) :- tlabel(I,(S,T),A), relevant(I,S).
f a_arity(A,0..max_action_arity) g = 1 :- action(A).
object(I,O) :- fval(I,(top,(O,)),1).

% Choose predicates from high-level language
f pred(F) : feature(F) g num_predicates.

% Tuples of variables/constants for lifted effects and preconditions
#defined constant/1.
argtuple(null_arg,0). % Explicit zero arity predicates
argtuple((C1,),1)     :- constant(C1).
argtuple((V1,),1)     :- V1 = 1..max_action_arity.
argtuple((C1,C2),2)   :- constant(C1), constant(C2).
argtuple((C1,V2),2)   :- constant(C1), V2 = 1..max_action_arity.
argtuple((V1,C2),2)   :- V1 = 1..max_action_arity, constant(C2).
argtuple((V1,V2),2)   :- V1 = 1..max_action_arity, V2 = 1..max_action_arity.

% Tuples of objects that ground the action schemas and atoms
objtuple(I,  null_arg,0)      :- instance(I). % Explicit zero arity predicates
objtuple(I,     (O1,),1)      :- object(I,O1), not constant(O1).
objtuple(I,   (O1,O2),2)      :- object(I,O1), object(I,O2), not constant(O1), not constant(O2), O1 ≠ O2.
objtuple(I,   (O1,O1),2)      :- object(I,O1), not constant(O1),                                 opt_equal_objects = 1.

% Tuples of objects/constants that appear as arguments to atoms
const_or_obj(I,O)             :- object(I,O).
const_or_obj(I,O)             :- instance(I), constant(O).
constobjtuple(I,  null_arg,0) :- instance(I). % Explicit zero arity predicates
constobjtuple(I,     (O1,),1) :- const_or_obj(I,O1).
constobjtuple(I,   (O1,O2),2) :- const_or_obj(I,O1), const_or_obj(I,O2), O1 ≠ O2.
constobjtuple(I,   (O1,O1),2) :- const_or_obj(I,O1),                     opt_equal_objects = 1.

% Assumption: predicates have arity < 3
:- p_arity(F,N), N > 2.

% Assert missing values for atoms (if some atom is not true, it is false)
fval(I,(F,OO),0)   :- feature(F),     f_static(I,F), p_arity(F,N), constobjtuple(I,OO,N),           not fval(I,(F,OO),1),   opt_fill = 1.
fval(I,(F,OO),S,0) :- feature(F), not f_static(I,F), p_arity(F,N), constobjtuple(I,OO,N), node(I,S), not fval(I,(F,OO),S,1), opt_fill = 1.

% Make sure we have full valuation of atoms
:-     f_static(I,F), p_arity(F,N), constobjtuple(I,OO,N),            f fval(I,(F,OO),0..1)   g ≠ 1, opt_fill = 1.
:- not f_static(I,F), p_arity(F,N), constobjtuple(I,OO,N), node(I,S), f fval(I,(F,OO),S,0..1) g ≠ 1, opt_fill = 1.

% Mapping of lifted arguments for atoms into grounded arguments. Lifted atom is pair (P,T) where P is
% predicate and T is tuple of variables and constants used to construct argument OO of grounded atom (P,OO).

% for nullary actions
map(I,(0,0,0,0),null_arg,null_arg,0)      :- instance(I).
map(I,(0,0,0,0),(C,),(C,),1)              :- constobjtuple(I,(C,),1), constant(C).
map(I,(0,0,0,0),(C1,C2),(C1,C2),2)        :- constobjtuple(I,(C1,C2),2), constant(C1), constant(C2).

% for unary actions
map(I,(O1,0,0,0),null_arg,null_arg,0)     :- objtuple(I,(O1,),1).
map(I,(O1,0,0,0),(C,),(C,),1)             :- objtuple(I,(O1,),1), constobjtuple(I,(C,),1), constant(C).
map(I,(O1,0,0,0),(1,),(O1,),1)            :- objtuple(I,(O1,),1).
map(I,(O1,0,0,0),(C1,C2),(C1,C2),2)       :- objtuple(I,(O1,),1), constobjtuple(I,(C1,C2),2), constant(C1), constant(C2).
map(I,(O1,0,0,0),(C,1),(C,O1),2)          :- objtuple(I,(O1,),1), constobjtuple(I,(C,O1),2), constant(C).
map(I,(O1,0,0,0),(1,C),(O1,C),2)          :- objtuple(I,(O1,),1), constobjtuple(I,(O1,C),2), constant(C).

% for actions of arity >= 2
map(I,(O1,O2,0,0),null_arg,null_arg,0)    :- objtuple(I,(O1,O2),2).
map(I,(O1,O2,0,0),(C,),(C,),1)            :- objtuple(I,(O1,O2),2), constobjtuple(I,(C,),1), constant(C).
map(I,(O1,O2,0,0),(1,),(O1,),1)           :- objtuple(I,(O1,O2),2).
map(I,(O1,O2,0,0),(2,),(O2,),1)           :- objtuple(I,(O1,O2),2).
map(I,(O1,O2,0,0),(C1,C2),(C1,C2),2)      :- objtuple(I,(O1,O2),2), constobjtuple(I,(C1,C2),2), constant(C1), constant(C2).
map(I,(O1,O2,0,0),(C,1),(C,O1),2)         :- objtuple(I,(O1,O2),2), constobjtuple(I,(C,O1),2), constant(C).
map(I,(O1,O2,0,0),(C,2),(C,O2),2)         :- objtuple(I,(O1,O2),2), constobjtuple(I,(C,O2),2), constant(C).
map(I,(O1,O2,0,0),(1,C),(O1,C),2)         :- objtuple(I,(O1,O2),2), constobjtuple(I,(O1,C),2), constant(C).
map(I,(O1,O2,0,0),(2,C),(O2,C),2)         :- objtuple(I,(O1,O2),2), constobjtuple(I,(O2,C),2), constant(C).
map(I,(O1,O2,0,0),(1,1),(O1,O1),2)        :- objtuple(I,(O1,O2),2), constobjtuple(I,(O1,O1),2).
map(I,(O1,O2,0,0),(1,2),(O1,O2),2)        :- objtuple(I,(O1,O2),2), constobjtuple(I,(O1,O2),2).
```

Figure 15: Listing of the ASP code (page 1/4)

```
map(I,(O1,O2,0,0),(2,1),(O2,O1),2)        :- objtuple(I,(O1,O2),2), constobjtuple(I,(O2,O1),2).
map(I,(O1,O2,0,0),(2,2),(O2,O2),2)        :- objtuple(I,(O1,O2),2), constobjtuple(I,(O2,O2),2).
map(I,(O1,0,O3,0),(3,),(O3,),1)           :- objtuple(I,(O1,O3),2).
map(I,(O1,0,O3,0),(C,3),(C,O3),2)         :- objtuple(I,(O1,O3),2), constobjtuple(I,(C,O3),2), constant(C).
map(I,(O1,0,O3,0),(3,C),(O3,C),2)         :- objtuple(I,(O1,O3),2), constobjtuple(I,(O3,C),2), constant(C).
map(I,(O1,0,O3,0),(1,3),(O1,O3),2)        :- objtuple(I,(O1,O3),2), constobjtuple(I,(O1,O3),2).
map(I,(O1,0,O3,0),(3,1),(O3,O1),2)        :- objtuple(I,(O1,O3),2), constobjtuple(I,(O3,O1),2).
map(I,(O1,0,O3,0),(3,3),(O3,O3),2)        :- objtuple(I,(O1,O3),2), constobjtuple(I,(O3,O3),2).
map(I,(O1,0,0,O4),(4,),(O4,),1)           :- objtuple(I,(O1,O4),2).
map(I,(O1,0,0,O4),(C,4),(C,O4),2)         :- objtuple(I,(O1,O4),2), constobjtuple(I,(C,O4),2), constant(C).
map(I,(O1,0,0,O4),(4,C),(O4,C),2)         :- objtuple(I,(O1,O4),2), constobjtuple(I,(O4,C),2), constant(C).
map(I,(O1,0,0,O4),(1,4),(O1,O4),2)        :- objtuple(I,(O1,O4),2), constobjtuple(I,(O1,O4),2).
map(I,(O1,0,0,O4),(4,1),(O4,O1),2)        :- objtuple(I,(O1,O4),2), constobjtuple(I,(O4,O1),2).
map(I,(O1,0,0,O4),(4,4),(O4,O4),2)        :- objtuple(I,(O1,O4),2), constobjtuple(I,(O4,O4),2).
map(I,(0,O2,O3,0),(2,3),(O2,O3),2)        :- objtuple(I,(O2,O3),2), constobjtuple(I,(O2,O3),2).
map(I,(0,O2,O3,0),(3,2),(O3,O2),2)        :- objtuple(I,(O2,O3),2), constobjtuple(I,(O3,O2),2).
map(I,(0,O2,0,O4),(2,4),(O2,O4),2)        :- objtuple(I,(O2,O4),2), constobjtuple(I,(O2,O4),2).
map(I,(0,O2,0,O4),(4,2),(O4,O2),2)        :- objtuple(I,(O2,O4),2), constobjtuple(I,(O4,O2),2).
map(I,(0,0,O3,O4),(3,4),(O3,O4),2)        :- objtuple(I,(O3,O4),2), constobjtuple(I,(O3,O4),2).
map(I,(0,0,O3,O4),(4,3),(O4,O3),2)        :- objtuple(I,(O3,O4),2), constobjtuple(I,(O4,O3),2).

map(I,(0,O2,0,0),(2,),(O2,),1)            :- objtuple(I,(O2,),1).
map(I,(0,0,O3,0),(3,),(O3,),1)            :- objtuple(I,(O3,),1).
map(I,(0,0,0,O4),(4,),(O4,),1)            :- objtuple(I,(O4,),1).


% Define variables used by actions, vars in tuples, and good arg tuples for actions
a_var(A,V)           :- a_arity(A,N), V = 1..N.
t_var((V,),V)        :- argtuple((V,),1),    not constant(V), V != null.
t_var((V1,V2),V1)    :- argtuple((V1,V2),2), not constant(V1).
t_var((V1,V2),V2)    :- argtuple((V1,V2),2), not constant(V2).
goodtuple(A,null_arg) :- action(A).
goodtuple(A,T)       :- action(A), argtuple(T,N), a_var(A,V) : t_var(T,V).

% Choice of lifted preconditions: prec(A,M,V) means atom M is prec of action A for V = 0 or 1
{ prec(A,(P,T),1) }      :- action(A), pred(P), p_arity(P,N), argtuple(T,N), goodtuple(A,T),
                            opt_allow_negative_precs = 0.

{ prec(A,(P,T),0..1) } 1 :- action(A), pred(P), p_arity(P,N), argtuple(T,N), goodtuple(A,T),
                            opt_allow_negative_precs = 1.


% Choice of lifted effects: eff(A,M,V) means atom M is effect of action A for V = 0 or 1
p_static(P) :- pred(P), f_static(I,P) : instance(I).
{ eff(A,(P,T),0..1) } 1 :- action(A), pred(P), not p_static(P), p_arity(P,N), argtuple(T,N), goodtuple(A,T).

% E1. Avoid noop actions and rule out contradictory effects
:- action(A), { eff(A,(P,T),0..1) : pred(P), p_arity(P,N), argtuple(T,N) } = 0.
:- eff(A,M,0), eff(A,M,1).

% Factored applicability relations (this implementation for action arities up to 4)

% Static predicates
fappl(I,A,null_arg,null_arg)   :- instance(I), action(A), a_arity(A,0),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(0,0,0,0),T,OO,K),       f_static(I,P).
fappl(I,A,(1,),(O1,))          :- instance(I), action(A), a_arity(A,1), objtuple(I,(O1,),1),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(O1,0,0,0),T,OO,K),      f_static(I,P).
fappl(I,A,(1,2),(O1,O2))       :- instance(I), action(A), a_arity(A,N), N >= 2, objtuple(I,(O1,O2),2),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(O1,O2,0,0),T,OO,K),     f_static(I,P).
fappl(I,A,(1,3),(O1,O3))       :- instance(I), action(A), a_arity(A,N), N >= 3, objtuple(I,(O1,O3),2),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(O1,0,O3,0),T,OO,K),     f_static(I,P).
fappl(I,A,(2,3),(O2,O3))       :- instance(I), action(A), a_arity(A,N), N >= 3, objtuple(I,(O2,O3),2),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(0,O2,O3,0),T,OO,K),     f_static(I,P).
fappl(I,A,(1,4),(O1,O4))       :- instance(I), action(A), a_arity(A,N), N >= 4, objtuple(I,(O1,O4),2),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(O1,0,0,O4),T,OO,K),     f_static(I,P).
fappl(I,A,(2,4),(O2,O4))       :- instance(I), action(A), a_arity(A,N), N >= 4, objtuple(I,(O2,O4),2),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(0,O2,0,O4),T,OO,K),     f_static(I,P).
fappl(I,A,(3,4),(O3,O4))       :- instance(I), action(A), a_arity(A,N), N >= 4, objtuple(I,(O3,O4),2),
                                  fval(I,(P,OO),V)   : prec(A,(P,T),V), map(I,(0,0,O3,O4),T,OO,K),     f_static(I,P).

% Dynamic predicates (value depends on states)
fappl(I,A,null_arg,null_arg,S) :- instance(I), action(A), a_arity(A,0), relevant(I,S),
                                  fappl(I,A,null_arg,null_arg),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(0,0,0,0),T,OO,K),   not f_static(I,P).
fappl(I,A,(1,),(O1,),S)        :- instance(I), action(A), a_arity(A,1), objtuple(I,(O1,),1), relevant(I,S),
                                  fappl(I,A,(1,),(O1,)),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(O1,0,0,0),T,OO,K),  not f_static(I,P).
fappl(I,A,(1,2),(O1,O2),S)     :- instance(I), action(A), a_arity(A,N), N >= 2, objtuple(I,(O1,O2),2), relevant(I,S),
                                  fappl(I,A,(1,2),(O1,O2)),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(O1,O2,0,0),T,OO,K), not f_static(I,P).
fappl(I,A,(1,3),(O1,O3),S)     :- instance(I), action(A), a_arity(A,N), N >= 3, objtuple(I,(O1,O3),2), relevant(I,S),
                                  fappl(I,A,(1,3),(O1,O3)),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(O1,0,O3,0),T,OO,K), not f_static(I,P).
fappl(I,A,(2,3),(O2,O3),S)     :- instance(I), action(A), a_arity(A,N), N >= 3, objtuple(I,(O2,O3),2), relevant(I,S),
                                  fappl(I,A,(2,3),(O2,O3)),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(0,O2,O3,0),T,OO,K), not f_static(I,P).
fappl(I,A,(1,4),(O1,O4),S)     :- instance(I), action(A), a_arity(A,N), N >= 4, objtuple(I,(O1,O4),2), relevant(I,S),
                                  fappl(I,A,(1,4),(O1,O4)),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(O1,0,0,O4),T,OO,K), not f_static(I,P).
fappl(I,A,(2,4),(O2,O4),S)     :- instance(I), action(A), a_arity(A,N), N >= 4, objtuple(I,(O2,O4),2), relevant(I,S),
                                  fappl(I,A,(2,4),(O2,O4)),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(0,O2,0,O4),T,OO,K), not f_static(I,P).
fappl(I,A,(3,4),(O3,O4),S)     :- instance(I), action(A), a_arity(A,N), N >= 4, objtuple(I,(O3,O4),2), relevant(I,S),
                                  fappl(I,A,(3,4),(O3,O4)),
                                  fval(I,(P,OO),S,V) : prec(A,(P,T),V), map(I,(0,0,O3,O4),T,OO,K), not f_static(I,P).
```

Figure 16: Listing of the ASP code (page 2/4)

```
% Factored next relation in the induced transition system

% Assumption in this implementation: each edge (S1,S2) labeled with unique action, and maps to unique grounded action A(OO)
:- tlabel(I,(S1,S2),A), tlabel(I,(S1,S2),B), A ≠ B.

% fnext(I,K,O,S1,S2) : for edge tlabel(I,(S1,S2),A), K-th argument of A(OO) is object O

% 1. Account for proper def of next(I,A,OO,S1,S2):
%   1.a. Choose next S2 if ground action A(OO) is applicable in S1
%   1.b. If A/k1 -> O1 in (S1,S2), then A/k2 -> O2, for some O2, for each arg k1 of action A in (S1,S2)
%        (i.e., A(OO) must be fully defined for (S1,S2))
%   1.c. It cannot be fnext(I,K,O1,S1,S2) and fnext(I,K,O2,S1,S2) with O1 ≠ O2
%   1.d. A(OO) cannot be mapped to two edges (S1,S2) and (S1,S3) with S2 ≠ S3
%   1.e. If tlabel(I,(S1,S2),A), then fnext(I,0,0,S1,S2) or fnext(I,1,_,S1,S2)

% 1.a. Choose next S2 if ground action A(OO) is applicable in S1
f fnext(I,0,0,  S1,S2) : tlabel(I,(S1,S2),A) g =1 :- action(A), a_arity(A,0), relevant(I,S1), fappl(I,A,null_arg,null_arg,S1).
f fnext(I,1,O1, S1,S2) : tlabel(I,(S1,S2),A) g =1 :- action(A), a_arity(A,1), relevant(I,S1), fappl(I,A,(1,),(O1,),S1).
1 f fnext(I,1,O1, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,2), relevant(I,S1), fappl(I,A,(1,2),(O1,O2),S1).
1 f fnext(I,2,O2, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,2), relevant(I,S1), fappl(I,A,(1,2),(O1,O2),S1).

1 f fnext(I,1,O1, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,3), relevant(I,S1),
                                                     fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(2,3),(O2,O3),S1).
1 f fnext(I,2,O2, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,3), relevant(I,S1),
                                                     fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(2,3),(O2,O3),S1).
1 f fnext(I,3,O3, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,3), relevant(I,S1),
                                                     fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(2,3),(O2,O3),S1).

1 f fnext(I,1,O1, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,4), relevant(I,S1),
                                                     fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(1,4),(O1,O4),S1),
                                                     fappl(I,A,(2,3),(O2,O3),S1), fappl(I,A,(2,4),(O2,O4),S1), fappl(I,A,(3,4),(O3,O4),S1).
1 f fnext(I,2,O2, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,4), relevant(I,S1),
                                                     fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(1,4),(O1,O4),S1),
                                                     fappl(I,A,(2,3),(O2,O3),S1), fappl(I,A,(2,4),(O2,O4),S1), fappl(I,A,(3,4),(O3,O4),S1).
1 f fnext(I,3,O3, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,4), relevant(I,S1),
                                                     fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(1,4),(O1,O4),S1),
                                                     fappl(I,A,(2,3),(O2,O3),S1), fappl(I,A,(2,4),(O2,O4),S1), fappl(I,A,(3,4),(O3,O4),S1).
1 f fnext(I,4,O4, S1,S2) : tlabel(I,(S1,S2),A) g  :- action(A), a_arity(A,4), relevant(I,S1),
                                                     fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(1,4),(O1,O4),S1),
                                                     fappl(I,A,(2,3),(O2,O3),S1), fappl(I,A,(2,4),(O2,O4),S1), fappl(I,A,(3,4),(O3,O4),S1).

% 1.b. If k1 -> O1 in (S1,S2), then k2 -> O2 in (S1,S2), for some O2, for each arg k1 of action A
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), fnext(I,K1,O1,S1,S2), K1 > 0, K2 = 1..N, K2 ≠ K1,
   not fnext(I,K2,O2,S1,S2) : object(I,O2).

% 1.c. It cannot be fnext(I,K,O1,S1,S2) and fnext(I,K,O2,S1,S2) with O1 < O2
:- relevant(I,S1), fnext(I,K,O1,S1,S2), fnext(I,K,O2,S1,S2), O1 < O2.

% 1.d. A(OO) cannot be mapped to two edges (S1,S2) and (S1,S3) with S2 < S3
:- relevant(I,S1), tlabel(I,(S1,S2),A), tlabel(I,(S1,S3),A), S2 < S3, a_arity(A,0).
:- relevant(I,S1), tlabel(I,(S1,S2),A), tlabel(I,(S1,S3),A), S2 < S3, a_arity(A,N), N >= 1,
   fnext(I,1,O,S1,S2), fnext(I,1,O,S1,S3), not diff_fnext(I,A,K2,S1,S2,S3) : K2 = 2..N.
diff_fnext(I,A,K,S1,S2,S3) :- relevant(I,S1), tlabel(I,(S1,S2),A), tlabel(I,(S1,S3),A), S2 < S3,
                              a_arity(A,N), N >= 1, K = 2..N, fnext(I,K,O1,S1,S2), fnext(I,K,O2,S1,S3), O1 ≠ O2.

% 1.e. If tlabel(I,(S1,S2),A), then fnext(I,0,0,S1,S2) or fnext(I,1,_,S1,S2)
:- relevant(I,S1), tlabel(I,(S1,S2),A), not fnext(I,0,0,S1,S2), not fnext(I,1,O,S1,S2) : object(I,O).

% 2. Check application of effects
%   2.a. If A(OO1) mapped to (S1,S2) and eff(A,(P,OO2),V), then fval(I,(P,OO2),S2,V)
%   2.b. If A(OO1) mapped to (S1,S2), fval(I,(P,OO2),S1,V) and fval(I,(P,OO2),S2,1-V), then eff(A,(P,OO2),1-V).

% 2.a. If A(OO1) mapped to (S1,S2) and eff(A,(P,OO2),V), then fval(I,(P,OO2),S2,V)
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,0),         fnext(I,0,0,S1,S2),                       eff(A,(P,T),V),
   map(I,(0,0,0,0),T,OO,K),   not f_static(I,P), fval(I,(P,OO),S2,1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,1),         fnext(I,1,O1,S1,S2),                      eff(A,(P,T),V),
   map(I,(O1,0,0,0),T,OO,K),  not f_static(I,P), fval(I,(P,OO),S2,1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 2, fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2), eff(A,(P,T),V),
   map(I,(O1,O2,0,0),T,OO,K), not f_static(I,P), fval(I,(P,OO),S2,1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 3, fnext(I,1,O1,S1,S2), fnext(I,3,O3,S1,S2), eff(A,(P,T),V),
   map(I,(O1,0,O3,0),T,OO,K), not f_static(I,P), fval(I,(P,OO),S2,1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 3, fnext(I,2,O2,S1,S2), fnext(I,3,O3,S1,S2), eff(A,(P,T),V),
   map(I,(0,O2,O3,0),T,OO,K), not f_static(I,P), fval(I,(P,OO),S2,1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 4, fnext(I,1,O1,S1,S2), fnext(I,4,O4,S1,S2), eff(A,(P,T),V),
   map(I,(O1,0,0,O4),T,OO,K), not f_static(I,P), fval(I,(P,OO),S2,1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 4, fnext(I,2,O2,S1,S2), fnext(I,4,O4,S1,S2), eff(A,(P,T),V),
   map(I,(0,O2,0,O4),T,OO,K), not f_static(I,P), fval(I,(P,OO),S2,1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 4, fnext(I,3,O3,S1,S2), fnext(I,4,O4,S1,S2), eff(A,(P,T),V),
   map(I,(0,0,O3,O4),T,OO,K), not f_static(I,P), fval(I,(P,OO),S2,1-V).

:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,0),         fnext(I,0,0,S1,S2),                       eff(A,(P,T),V),
   map(I,(0,0,0,0),T,OO,K),       f_static(I,P), fval(I,(P,OO),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,1),         fnext(I,1,O1,S1,S2),                      eff(A,(P,T),V),
   map(I,(O1,0,0,0),T,OO,K),      f_static(I,P), fval(I,(P,OO),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 2, fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2), eff(A,(P,T),V),
   map(I,(O1,O2,0,0),T,OO,K),     f_static(I,P), fval(I,(P,OO),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 3, fnext(I,1,O1,S1,S2), fnext(I,3,O3,S1,S2), eff(A,(P,T),V),
   map(I,(O1,0,O3,0),T,OO,K),     f_static(I,P), fval(I,(P,OO),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 3, fnext(I,2,O2,S1,S2), fnext(I,3,O3,S1,S2), eff(A,(P,T),V),
   map(I,(0,O2,O3,0),T,OO,K),     f_static(I,P), fval(I,(P,OO),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 4, fnext(I,1,O1,S1,S2), fnext(I,4,O4,S1,S2), eff(A,(P,T),V),
   map(I,(O1,0,0,O4),T,OO,K),     f_static(I,P), fval(I,(P,OO),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 4, fnext(I,2,O2,S1,S2), fnext(I,4,O4,S1,S2), eff(A,(P,T),V),
   map(I,(0,O2,0,O4),T,OO,K),     f_static(I,P), fval(I,(P,OO),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N), N >= 4, fnext(I,3,O3,S1,S2), fnext(I,4,O4,S1,S2), eff(A,(P,T),V),
   map(I,(0,0,O3,O4),T,OO,K),     f_static(I,P), fval(I,(P,OO),1-V).
```

Figure 17: Listing of the ASP code (page 3/4)

```
% 2.b. If A(OO1) mapped to (S1,S2), fval(I,(P,OO2),S1,V) and fval(I,(P,OO2),S2,1-V), then eff(A,(P,OO2),1-V).
:- relevant(I,S1), tlabel(I,(S1,S2),A),
   pred(P), fval(I,(P,null_arg),S1,V), fval(I,(P,null_arg),S2,1-V),
   not eff(A,(P,null_arg),1-V).

:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N),
   pred(P), fval(I,(P,OO),S1,V), fval(I,(P,OO),S2,1-V), constobjtuple(I,OO,1),
   not eff(A,(P,T),1-V) : map(I,(0,0,0,0),T,OO,1),                        N >= 0;
   not eff(A,(P,T),1-V) : map(I,(O1,0,0,0),T,OO,1),  fnext(I,1,O1,S1,S2), N >= 1;
   not eff(A,(P,T),1-V) : map(I,(0,O2,0,0),T,OO,1),  fnext(I,2,O2,S1,S2), N >= 2;
   not eff(A,(P,T),1-V) : map(I,(0,0,O3,0),T,OO,1),  fnext(I,3,O3,S1,S2), N >= 3;
   not eff(A,(P,T),1-V) : map(I,(0,0,0,O4),T,OO,1),  fnext(I,4,O4,S1,S2), N >= 4.

:- relevant(I,S1), tlabel(I,(S1,S2),A), a_arity(A,N),
   pred(P), fval(I,(P,OO),S1,V), fval(I,(P,OO),S2,1-V), constobjtuple(I,OO,2),
   not eff(A,(P,T),1-V) : map(I,(0,0,0,0),T,OO,2),                                             N >= 0;
   not eff(A,(P,T),1-V) : map(I,(O1,0,0,0),T,OO,2),  fnext(I,1,O1,S1,S2),                      N >= 1;
   not eff(A,(P,T),1-V) : map(I,(O1,O2,0,0),T,OO,2), fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2), N >= 2;
   not eff(A,(P,T),1-V) : map(I,(O1,0,O3,0),T,OO,2), fnext(I,1,O1,S1,S2), fnext(I,3,O3,S1,S2), N >= 3;
   not eff(A,(P,T),1-V) : map(I,(0,O2,O3,0),T,OO,2), fnext(I,2,O2,S1,S2), fnext(I,3,O3,S1,S2), N >= 3;
   not eff(A,(P,T),1-V) : map(I,(O1,0,0,O4),T,OO,2), fnext(I,1,O1,S1,S2), fnext(I,4,O4,S1,S2), N >= 4;
   not eff(A,(P,T),1-V) : map(I,(0,O2,0,O4),T,OO,2), fnext(I,2,O2,S1,S2), fnext(I,4,O4,S1,S2), N >= 4;
   not eff(A,(P,T),1-V) : map(I,(0,0,O3,O4),T,OO,2), fnext(I,3,O3,S1,S2), fnext(I,4,O4,S1,S2), N >= 4.

% 3. Check application of applicable ground actions
:- a_arity(A,2), fappl(I,A,(1,2),(O1,O2),S1), not fnext(I,2,O2,S1,S2) : fnext(I,1,O1,S1,S2).

:- a_arity(A,3), fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(2,3),(O2,O3),S1),
   not fnext(I,2,O2,S1,S2) : fnext(I,1,O1,S1,S2).
:- a_arity(A,3), fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(2,3),(O2,O3),S1),
   not fnext(I,3,O3,S1,S2) : fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2).

:- a_arity(A,4), fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(1,4),(O1,O4),S1),
   fappl(I,A,(2,3),(O2,O3),S1), fappl(I,A,(2,4),(O2,O4),S1), fappl(I,A,(3,4),(O3,O4),S1),
   not fnext(I,2,O2,S1,S2) : fnext(I,1,O1,S1,S2).
:- a_arity(A,4), fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(1,4),(O1,O4),S1),
   fappl(I,A,(2,3),(O2,O3),S1), fappl(I,A,(2,4),(O2,O4),S1), fappl(I,A,(3,4),(O3,O4),S1),
   not fnext(I,3,O3,S1,S2) : fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2).
:- a_arity(A,4), fappl(I,A,(1,2),(O1,O2),S1), fappl(I,A,(1,3),(O1,O3),S1), fappl(I,A,(1,4),(O1,O4),S1),
   fappl(I,A,(2,3),(O2,O3),S1), fappl(I,A,(2,4),(O2,O4),S1), fappl(I,A,(3,4),(O3,O4),S1),
   not fnext(I,4,O4,S1,S2) : fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2), fnext(I,3,O3,S1,S2).

% Fundamental constraints

% C1. Different nodes are different states with respect to chosen predicates
:- relevant(I,S1), relevant(I,S2), S1 < S2, fval(I,(P,OO),S2,V) : fval(I,(P,OO),S1,V), pred(P), not f_static(I,P).

% C2. If (S1,S2) has label A, then A(OO) maps S1 to S2 for some object tuple OO
:- relevant(I,S1), a_arity(A,0), tlabel(I,(S1,S2),A), not fnext(I,0,0,S1,S2).
:- relevant(I,S1), a_arity(A,1), tlabel(I,(S1,S2),A), #count {    O1 : fnext(I,1,O1,S1,S2) } != 1.
:- relevant(I,S1), a_arity(A,2), tlabel(I,(S1,S2),A), #count { O1,O2 : fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2) } != 1.
:- relevant(I,S1), a_arity(A,3), tlabel(I,(S1,S2),A),
   #count {    O1,O2,O3 : fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2), fnext(I,3,O3,S1,S2) } != 1.
:- relevant(I,S1), a_arity(A,4), tlabel(I,(S1,S2),A),
   #count { O1,O2,O3,O4 : fnext(I,1,O1,S1,S2), fnext(I,2,O2,S1,S2), fnext(I,3,O3,S1,S2), fnext(I,4,O4,S1,S2) } != 1.

% Break symmetries (taken from STRIPS learner's ASP code)
a_atom(1,A,M) :- prec(A,M,V).
a_atom(2,A,M) :-  eff(A,M,V).
a_atom(A,M)   :- a_atom(I,A,M), I = 1..2.

:- V = 1..max_action_arity-1, action(A),
   1 #sum{ -1,I,P,VV,1 : a_atom(I,A,(P,(  V, VV)));
           -1,I,P,VV,2 : a_atom(I,A,(P,( VV,  V)));
            1,I,P,VV,1 : a_atom(I,A,(P,(V+1, VV)));
            1,I,P,VV,2 : a_atom(I,A,(P,( VV,V+1))) },
   opt_symmetries = 1.

% Optimization (lexicographic ordering)
% - Prefer models with minimum sum of actions' cardinalities
% - Prefer models with minimum sum of (non-static) predicates' cardinalities
% - Prefer models with minimum sum of (static) predicates' cardinalities
% - Prefer models with minimum number of effects
% - Prefer models with minimum number of preconditions
#minimize { 1+N@10, A : a_arity(A,N) }.
#minimize { 1+N@8, P : pred(P), p_arity(P,N), not p_static(P) }.
#minimize { 1+N@6, P : pred(P), p_arity(P,N), p_static(P) }.
#minimize { 1@4, A, P, T, V : eff(A,(P,T),V) }.
#minimize { 1@2, A, P, T, V : prec(A,(P,T),V) }.

% Default is to display nothing
#show.

% Display objects and constants
#show object/2.
#show constant/1.

% Display selected predicates
#show pred/1.
#show p_static(P) : p_static(P), pred(P).

% Display action schemas
#show action/1.
#show a_arity/2.
#show prec/3.
#show eff/3.
```

Figure 18: Listing of the ASP code (page 4/4)