Learning Action Models from State Observations

#1186

Abstract

This paper presents a classical planning compilation for learning STRIPS action models from state observations. The compilation approach does not require observing the precise actions that produced the observations because such actions are determined by a planner. Furthermore, the presented compilation is extensible to assess how well a STRIPS action model matches a given set of observations. Last but not least, the paper evaluates the performance of the proposed approach by learning action models for a wide range of classical planning domains from the International Planning Competition and assessing the learned models with respect to (1) the corresponding reference models and (2), given observations test sets.

1 Introduction

Besides *plan synthesis* [Ghallab *et al.*, 2004], planning action models are also useful for *plan/goal recognition* [Ramírez, 2012]. At these planning tasks, automated planners are required to reason about an action model that correctly and completely captures the possible world transitions [Geffner and Bonet, 2013]. Unfortunately, building planning action models is complex, even for planning experts, and this knowledge acquisition task is a bottleneck that limits the potential of *AI planning* [Kambhampati, 2007].

The Machine Learning of planning action models is a promising alternative to hand-coding them and nowadays, there exist sophisticated algorithms like AMAN [Zhuo and Kambhampati, 2013], ARMS [Yang *et al.*, 2007], LOCM [Cresswell *et al.*, 2013] or SLAF [Amir and Chang, 2008]. Motivated by recent advances on the synthesis of different kinds of generative models with classical planning [Bonet *et al.*, 2009; Segovia-Aguas *et al.*, 2016; 2017], this paper presents a novel approach for learning STRIPS action models that introduces the following contributions:

1. *Learning action models as planning*. The practicality of the compilation approach allow us to report results over a wide range of planning domains. Furthermore, opens up a way towards the *bootstrapping* of planning action models (a planner exploring its state space and using the obtained data to learn/update its action model).
2. *Learning from state observations*. Our approach does not require the learning samples to be observations of actions but simply state observations, broadening the range of application to external observers.
3. *Model evaluation*. The compilation is extensible to assess how well a STRIPS action model matches a given set of observations, which enables *model recognition*.

2 Background

Our approach for learning STRIPS action models is compiling this leaning task into a classical planning task with conditional effects.

Classical planning with conditional effects

We use F to denote the set of *fluents* (propositional variables) describing a state. A *literal* l is a valuation of a fluent $f \in F$, i.e. either $l = f$ or $l = \neg f$. A set of literals L represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). We use $\mathcal{L}(F)$ to denote the set of all literal sets on F, i.e. all partial assignments of values to fluents.

A *state* s is a full assignment of values to fluents, i.e. $|s| = |F|$, so the size of the state space is $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions but often, we will abuse notation by defining a state s only in terms of the fluents that are true in s, as is common in STRIPS planning.

A *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions. Each action $a \in A$ comprises three sets of literals:

- $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, called *preconditions*, the literals that must hold for the action $a \in A$ to be applicable.
- $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, called *positive effects*, that defines the fluents set to true by the application of the action $a \in A$.
- $\mathsf{eff}^-(a) \subseteq \mathcal{L}(F)$, called *negative effects*, that defines the fluents set to false by the action application.

We say that an action $a \in A$ is *applicable* in a state s iff $\mathsf{pre}(a) \subseteq s$. The result of applying a in s is the *successor state* denoted by $\theta(s, a) = \{s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)\}$.

A *classical planning problem* is a tuple $P = \langle F, A, I, G \rangle$, where I is an initial state and $G \subseteq \mathcal{L}(F)$ is a goal condition. A *plan* for P is an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ that

induces the *state trajectory* $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and, for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The *plan length* is denoted with $|\pi| = n$. A plan π *solves* P iff $G \subseteq s_n$, i.e. if the goal condition is satisfied at the last state reached after following the application of the plan π in the initial state I.

An action $a \in A$ with conditional effects is defined as a set of *preconditions* $\mathsf{pre}(a) \in \mathcal{L}(F)$ and a set of *conditional effects* $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two sets of literals $C \in \mathcal{L}(F)$, the *condition*, and $E \in \mathcal{L}(F)$, the *effect*. An action $a \in A$ is *applicable* in a state s if and only if $\mathsf{pre}(a) \subseteq s$, and the *triggered effects* resulting from the action application are the effects whose conditions hold in s:

$$triggered(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

The result of applying action a in state s is the *successor* state $\theta(s, a) = \{s \setminus \mathsf{eff}_c^-(s, a)) \cup \mathsf{eff}_c^+(s, a)\}$ where $\mathsf{eff}_c^-(s, a) \subseteq triggered(s, a)$ and $\mathsf{eff}_c^+(s, a) \subseteq triggered(s, a)$ are, respectively, the triggered *negative* and *positive* effects.

2.1 STRIPS action schemes

This work addresses the learning of PDDL action schemes that follow the STRIPS requirement [McDermott *et al.*, 1998; Fox and Long, 2003]. Figure 1 shows the *stack* action schema, coded in PDDL, from a four-operator *blocksworld* [Slaney and Thiébaux, 2001].

To formalize the output of the learning task, we assume that fluents F are instantiated from a set of *predicates* Ψ, as in PDDL. Each predicate $p \in \Psi$ has an argument list of arity $ar(p)$. Given a set of *objects* Ω, the set of fluents F is induced by assigning objects in Ω to the arguments of predicates in Ψ, i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ s.t. Ω^k is the k-th Cartesian power of Ω.

Let $\Omega_v = \{v_i\}_{i=1}^{\max_{a \in A} ar(a)}$ be a new set of objects ($\Omega \cap \Omega_v = \emptyset$), denoted as *variable names*, and that is bound by the maximum arity of an action in a given planning frame. For instance, in a three-block blocksworld $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the operators with the maximum arity, `stack` and `unstack`, have arity two.

Let us define F_v, a new set of fluents $F \cap F_v = \emptyset$, that results from instantiating Ψ using only the objects in Ω_v and that defines the elements that can appear in an action schema. For the blocksworld, F_v={`handempty`, `holding(`v_1`)`, `holding(`v_2`)`, `clear(`v_1`)`, `clear(`v_2`)`, `ontable(`v_1`)`, `ontable(`v_2`)`, `on(`v_1, v_1`)`, `on(`v_1, v_2`)`, `on(`v_2, v_1`)`, `on(`v_2, v_2`)`}.

We assume also that actions $a \in A$ are instantiated from STRIPS operator schemes $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$ where:

- $head(\xi) = \langle name(\xi), pars(\xi) \rangle$, is the operator *header* defined by its name and the corresponding *variable names*, $pars(\xi) = \{v_i\}_{i=1}^{ar(\xi)}$. The headers of a four-operator blocksworld are `pickup(`v_1`)`, `putdown(`v_1`)`, `stack(`v_1, v_2`)` and `unstack(`v_1, v_2`)`.

```
(:action stack
 :parameters (?v1 ?v2 - object)
 :precondition (and (holding ?v1) (clear ?v2))
 :effect (and (not (holding ?v1))
              (not (clear ?v2))
              (handempty) (clear ?v1)
              (on ?v1 ?v2)))
```

Figure 1: STRIPS operator schema coding, in PDDL, the *stack* action from a four-operator *blocksworld*.

- The preconditions $pre(\xi) \subseteq F_v$, the negative effects $del(\xi) \subseteq F_v$, and the positive effects $add(\xi) \subseteq F_v$ such that, $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$.

Finally, we define $F_v(\xi) \subseteq F_v$ as the subset of elements that can appear in a given action schema ξ and that confine the space of possible action models. For instance, for the *stack* action schema F_v(`stack`) $= F_v$ while F_v(`pickup`)={`handempty`, `holding(`v_1`)`, `clear(`v_1`)`, `ontable(`v_1`)`, `on(`v_1, v_1`)`} only contains the fluents from F_v that do not involve v_2 because the action header contains the single parameter v_1.

3 Learning STRIPS action models

Learning STRIPS action models from plans where every action in the plan is available as well as its corresponding *pre-* and *post-states*, is straightforward. In this case, STRIPS operator schemes are derived lifting the literals that change between the pre and post-state of the corresponding action executions. Preconditions are derived lifting the minimal set of literals that appears in all the corresponding pre-states.

This paper addresses a more challenging learning task, where less input knowledge is available. The addressed learning task corresponds to observing an agent acting in the world but watching only the results of its plan executions, the actual executed actions are unobserved. This learning task is defined as $\Lambda = \langle \Xi, \Psi, \mathcal{O} \rangle$, where:

- Ξ is the set of empty operator schemes, wherein each $\xi \in \Xi$ is only composed of $head(\xi)$.
- Ψ is the set of predicates, that define the abstract state space of a given classical planning frame.
- $\mathcal{O} = \langle s_0, s_1, \ldots, s_n \rangle$ is a sequence of *state observations* obtained observing the execution of an *unobserved* plan $\pi = \langle a_1, \ldots, a_n \rangle$.

A solution to Λ is a set of operator schema Ξ' compliant with the headers Ξ, the predicates Ψ, and the given sequence of state observations $\mathcal{O}$. A planning compilation is a suitable approach for addressing a Λ learning task because a solution must not only determine the STRIPS action model Ξ' but also, the *unobserved* plan $\pi = \langle a_1, \ldots, a_n \rangle$, that explains $\mathcal{O}$. Figure 2 shows a Λ task for learning a STRIPS action model in the blocksworld from the sequence of five state observations that corresponds to inverting a 2-blocks tower.

3.1 Learning with classical planning

Our approach for addressing the learning task is compiling Λ into a classical planning task P_Λ with conditional effects.

```
;;;;;; Headers in Ξ
(pickup v1) (putdown v1)
(stack v1 v2} (unstack v1 v2)

;;;;;; Predicates Ψ
(handempty) (holding ?o  - object)
(clear ?o - object) (ontable ?o - object)
(on ?o1 - object ?o2 - object)

;;;;;; Observations O
;;; observation #0
(clear block2) (on block2 block1)
(ontable block1) (handempty)

;;; observation #1
(holding block2) (clear block1) (ontable block1)

;;; observation #2
(clear block1) (ontable block1)
(clear block2) (ontable block2) (handempty)

;;; observation #3
(holding block1) (clear block2) (ontable block2)

;;; observation #4
(clear block1) (on block1 block2)
(ontable block2) (handempty)
```

Figure 2: Example of a Λ task for learning a STRIPS action model in the *blocksworld* from a sequence of five state observations.

The intuition behind the compilation is that a solution to the resulting classical planning task is a sequence of actions that:

1. **Programs the STRIPS action model Ξ'.** A solution plan starts with a *prefix* that, for each $\xi \in \Xi$, determines which fluents $f \in F_v(\xi)$ belong to its $pre(\xi)$, $del(\xi)$ and $add(\xi)$ sets.
2. **Validates the STRIPS action model Ξ' in $\mathcal{O}$.** The solution plan continues with a postfix that produces the given sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ using the programmed action model Ξ'.

Given a learning task $\Lambda = \langle \Xi, \Psi, \mathcal{O} \rangle$ the compilation outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$:

- F_Λ contains:
  - The set of fluents F built instantiating the predicates Ψ with the objects appearing in the input observations $\mathcal{O}$.
  - Fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_v(\xi)$, that represent the programmed action model. If a fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative effect/positive effect in the STRIPS operator schema $\xi \in \Xi'$. For instance, the preconditions of the *stack* schema (Figure 1) is represented by the pair of fluents `pre_holding_stack_`v_1 and `pre_clear_stack_`v_2 set to *True*.
  - Fluent $mode_{prog}$ indicating whether the operator schemes are programmed or validated (already programmed) and fluents $\{test_i\}_{1 \leq i \leq n}$, indicating the observation where the action model is validated.
- I_Λ contains the fluents from F that encode s_0 (the first observation) and $mode_{prog}$ set to true. Our compilation assumes that initially operator schemes are programmed with every possible precondition, no negative effect and no positive effect. Therefore fluents $pre_f(\xi)$, for every $f \in F_v(\xi)$, hold also at the initial state.
- $G_\Lambda = \bigcup_{1 \leq i \leq n} \{test_i\}$, indicates that the programmed action model is validated in all the input observations.
- A_Λ comprises three kinds of actions:
  1. Actions for *programming* operator schema $\xi \in \Xi$:
     - Actions for **removing** a *precondition* $f \in F_v(\xi)$ from the action schema $\xi \in \Xi$.

$$\begin{aligned} \mathsf{pre}(\mathsf{programPre}_{\mathsf{f},\xi}) =& \{\neg del_f(\xi), \neg add_f(\xi), \\ & mode_{prog}, pre_f(\xi)\}, \\ \mathsf{cond}(\mathsf{programPre}_{\mathsf{f},\xi}) =& \{\emptyset\} \rhd \{\neg pre_f(\xi)\}. \end{aligned}$$

     - Actions for **adding** a *negative* or *positive* effect $f \in F_v(\xi)$ to the action schema $\xi \in \Xi$.

$$\begin{aligned} \mathsf{pre}(\mathsf{programEff}_{\mathsf{f},\xi}) =& \{\neg del_f(\xi), \neg add_f(\xi), \\ & mode_{prog}\}, \\ \mathsf{cond}(\mathsf{programEff}_{\mathsf{f},\xi}) =& \{pre_f(\xi)\} \rhd \{del_f(\xi)\}, \\ & \{\neg pre_f(\xi)\} \rhd \{add_f(\xi)\}. \end{aligned}$$

  2. Actions for *applying* an already programmed operator schema $\xi \in \Xi$ bound with the objects $\omega \subseteq \Omega^{ar(\xi)}$. Given that the operators headers are known, the variables $pars(\xi)$ are bound to the objects in ω appearing at the same position. Figure 3 shows the PDDL encoding of the action for applying a programmed operator $stack$.

$$\begin{aligned} \mathsf{pre}(\mathsf{apply}_{\xi,\omega}) =& \{pre_f(\xi) \implies p(\omega)\}_{\forall p \in \Psi, f = p(pars(\xi))}, \\ \mathsf{cond}(\mathsf{apply}_{\xi,\omega}) =& \{del_f(\xi)\} \rhd \{\neg p(\omega)\}_{\forall p \in \Psi, f = p(pars(\xi))}, \\ & \{add_f(\xi)\} \rhd \{p(\omega)\}_{\forall p \in \Psi, f = p(pars(\xi))}, \\ & \{mode_{prog}\} \rhd \{\neg mode_{prog}\}. \end{aligned}$$

  3. Actions for *validating* an observation $1 \leq i \leq n$.

$$\begin{aligned} \mathsf{pre}(\mathsf{validate}_{\mathsf{i}}) =& s_i \cup \{test_j\}_{j \in 1 \leq j < i} \\ & \cup \{\neg test_j\}_{j \in i \leq j \leq n} \cup \{\neg mode_{prog}\}, \\ \mathsf{cond}(\mathsf{validate}_{\mathsf{i}}) =& \{\emptyset\} \rhd \{test_i\}. \end{aligned}$$

3.2 Compilation properties

Lemma 1. *Soundness. Any classical plan π that solves P_Λ induces an action model Ξ' that solves the Λ learning task.*

Proof sketch. The compilation forces that once the preconditions of an operator schema $\xi \in \Xi'$ are programmed, they cannot be altered. The same happens with the positive and negative effects. Furthermore, once operator schemes Ξ' are programmed, they can only

```
(:action apply_stack
  :parameters (?o1 - object ?o2 - object)
  :precondition
   (and (or (not (pre_on_stack_v1_v1)) (on ?o1 ?o1))
        (or (not (pre_on_stack_v1_v2)) (on ?o1 ?o2))
        (or (not (pre_on_stack_v2_v1)) (on ?o2 ?o1))
        (or (not (pre_on_stack_v2_v2)) (on ?o2 ?o2))
        (or (not (pre_ontable_stack_v1)) (ontable ?o1))
        (or (not (pre_ontable_stack_v2)) (ontable ?o2))
        (or (not (pre_clear_stack_v1)) (clear ?o1))
        (or (not (pre_clear_stack_v2)) (clear ?o2))
        (or (not (pre_holding_stack_v1)) (holding ?o1))
        (or (not (pre_holding_stack_v2)) (holding ?o2))
        (or (not (pre_handempty_stack)) (handempty)))
  :effect
   (and (when (del_on_stack_v1_v1) (not (on ?o1 ?o1)))
        (when (del_on_stack_v1_v2) (not (on ?o1 ?o2)))
        (when (del_on_stack_v2_v1) (not (on ?o2 ?o1)))
        (when (del_on_stack_v2_v2) (not (on ?o2 ?o2)))
        (when (del_ontable_stack_v1) (not (ontable ?o1)))
        (when (del_ontable_stack_v2) (not (ontable ?o2)))
        (when (del_clear_stack_v1) (not (clear ?o1)))
        (when (del_clear_stack_v2) (not (clear ?o2)))
        (when (del_holding_stack_v1) (not (holding ?o1)))
        (when (del_holding_stack_v2) (not (holding ?o2)))
        (when (del_handempty_stack) (not (handempty)))
        (when (add_on_stack_v1_v1) (on ?o1 ?o1))
        (when (add_on_stack_v1_v2) (on ?o1 ?o2))
        (when (add_on_stack_v2_v1) (on ?o2 ?o1))
        (when (add_on_stack_v2_v2) (on ?o2 ?o2))
        (when (add_ontable_stack_v1) (ontable ?o1))
        (when (add_ontable_stack_v2) (ontable ?o2))
        (when (add_clear_stack_v1) (clear ?o1))
        (when (add_clear_stack_v2) (clear ?o2))
        (when (add_holding_stack_v1) (holding ?o1))
        (when (add_holding_stack_v2) (holding ?o2))
        (when (add_handempty_stack) (handempty))
        (when (modeProg) (not (modeProg)))))
```

Figure 3: Action for applying an already programmed schema $stack$ as encoded in PDDL (implications are coded as disjunctions).

be applied because of the $mode_{prog}$ fluent. Finally, P_Λ is only solvable if fluents $\{test_i\}$, $1 \leq i \leq n$ hold at the last reached state. These goals can only be achieved executing an applicable sequence of programmed operator schemes that reaches every state $s_i \in \mathcal{O}$, starting from s_0 and following the sequence $1 \leq i \leq n$ which means that the programmed action model Ξ' complies with the provided observations and hence, solves Λ. □

Lemma 2. *Completeness. Any* STRIPS *action model Ξ' that solves a $\Lambda = \langle \Psi, \Xi, \mathcal{O} \rangle$ learning task, is computable solving the corresponding classical planning task P_Λ.*

Proof sketch. The compilation does not discard any possible STRIPS action schema definable within F_v that satisfy the state trajectory constraints given by the $\mathcal{O}$ sequence. By definition, $F_v(\xi) \subseteq F_\Lambda$ fully captures the full set of elements that can appear in a STRIPS action schema $\xi \in \Xi$ given its header and the set of predicates Ψ. □

The size of the compiled classical planning instances depends on the number of input examples. The larger the number of examples, the larger the compilation (a larger number of "test" fluents and "validate" actions), and also the more actions the solution plan requires to derive the action model.

4 Evaluation of STRIPS action models

We assess how well a STRIPS action model Ξ explains a given sequence of observations $\mathcal{O}$ according to the amount of *edition* that is required by Ξ to produce $\mathcal{O}$. In the extreme, if Ξ perfectly explains $\mathcal{O}$, no model *edition* is necessary.

4.1 Edition of STRIPS action models

Let us define first the *operations* allowed to edit a given STRIPS action model. With the aim of keeping tractable the branching factor of the planning instance that results from our compilation, we only define two *edit operations*:

- *Deletion.* A fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$, such that $f \in F_v(\xi)$, is removed from the operator schema $\xi \in \Xi$.
- *Insertion.* A fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$, such that $f \in F_v(\xi)$, is added to the operator schema $\xi \in \Xi$.

We can now formalize an edit distance that quantifies how dissimilar two given STRIPS action models are. The distance is symmetric and meets the *metric axioms* provided that the two STRIPS *edit operations* have the same positive cost.

Definition 3. *Let Ξ and Ξ' be two* STRIPS *action models, both built from the same set of possible elements F_v. The* **edit distance***, denoted as $\delta(\Xi, \Xi')$, is the minimum number of* edit operations *required to transform Ξ into Ξ'.*

Since F_v is a bound set, the maximum number of edits that can be introduced to a given action model defined within F_v is bound as well. In more detail, for an operator schema $\xi \in \Xi$ the maximum number of edits that can be introduced to their precondition set is $|F_v(\xi)|$ while the max number of edits that can be introduced to the effects are two times $|F_v(\xi)|$.

Definition 4. *The* ***maximum edit distance*** *of an* STRIPS *action model Ξ built from the set of possible elements F_v is $\delta(\Xi, *) = \sum_{\xi \in \Xi} 3|F_v(\xi)|$ and sets an upperbound on the distance from Ξ to any other* STRIPS *action model definable within F_v.*

4.2 The observation edit distance

We define now an edit distance to asses the quality of learned models with respect to a sequence of observations (that acts as a test set).

Definition 5. *Given an action model Ξ built from the set of possible elements F_v and a sequence of observations $\mathcal{O}$. The* **observation edit distance***, denoted by $\delta(\Xi, \mathcal{O})$, is the minimal edit distance from Ξ to any model Ξ' such that: (1) Ξ' is also definable within F_v and (2), Ξ' can produce a plan $\pi = \langle a_1, \ldots, a_n \rangle$ that induces $\mathcal{O} = \langle s_0, s_1, \ldots, s_n \rangle$, the given observations sequence; i.e.,*

$$\delta(\Xi, \mathcal{O}) = \min_{\forall \Xi' \rightarrow \mathcal{O}} \delta(\Xi, \Xi')$$

Because its semantic nature, the *observation edit distance* is robust to learning episodes where actions are *reformulated* and still compliant with the learning inputs. For instance a learning episode can interchange the roles of operators with matching headers, or the roles of action parameters with matching types (e.g. the *blocksworld* operator `stack` can be *learned* with the preconditions and effects of the `unstack` operator and vice versa).

Furthermore, the *observation edit distance* enables the recognition of STRIPS action models. The idea, taken from *plan recognition as planning* [Ramırez and Geffner, 2009], is

to map distances into likelihoods. We can map the *observation edit distance* into a likelihood with the following expression $P(\mathcal{O}|\Xi) = 1 - \frac{\delta(\Xi,\mathcal{O})}{\delta(\Xi,*)}$. The larger the *observation edit distance* the lower this likelihood.

4.3 Computing the observation edit distance

Our compilation is extensible to compute the *observations edit distance*. A solution to the planning task resulting from the extended compilation is a sequence of actions that:

1. **Edits the STRIPS action model Ξ.** A solution plan starts with a *prefix* that edits Ξ.
2. **Validates the edited model Ξ in $\mathcal{O}$.** The solution plan continues with a postfix that validates the edited model on the given observations $\mathcal{O}$.

In this case the tuple $\Lambda = \langle \Xi, \Psi, \mathcal{O} \rangle$ does not represent a learning task but the task of editing the STRIPS action model Ξ to produce the observations $\mathcal{O}$. The extended compilation outputs a classical planning task $P'_\Lambda = \langle F_\Lambda, A'_\Lambda, I'_\Lambda, G_\Lambda \rangle$:

- F_Λ and G_Λ are defined as in the previous compilation.
- I'_Λ contains the fluents from F that encode s_0 (the first observation) and $mode_{prog}$ set to true. In addition, the given STRIPS action model Ξ is now encoded in the initial state. This means that the fluents $pre_f(\xi)/del_f(\xi)/add_f(\xi)$, with $f \in F_v(\xi)$, hold in the initial state iff they appear in the given Ξ model.
- A'_Λ, comprises the same three kinds of actions of A_Λ. The actions for *applying* an already programmed operator scheme and the actions for *validating* an observation are defined exactly as in the previous compilation. The only difference here is the actions for *programming* operator schemes $\xi \in \Xi$ that now implement the two *edit operations* (i.e. include actions for adding a precondition and for removing a negative/positive effect).

Since we are computing the *observations edit distance*, we are not interested in the resulting edited model but in the number of *edit operations* required to produce a model validable in the given observations. The *observation edit distance* is exactly computed if P'_Λ is optimally solved (according to the number of edit actions), and is approximate if P'_Λ is solved with a satisfying planner or furthermore, if what is solved is not P'_Λ but its relaxation, e.g. the *delete relaxation* [Bonet and Geffner, 2001].

5 Evaluation

This section reports a two-fold evaluation of the learned models: (1) with respect to a given observation test set and (2), with respect to a reference model.

Reproducibility

We used 15 IPC domains that satisfy the STRIPS requirement [Fox and Long, 2003], taken from the PLANNING.DOMAINS repository [Muise, 2016]. For the learning of the STRIPS action models we only used observations sequences $\langle s_0, s_1, \ldots, s_{24} \rangle$ of twenty five states per domain. The set of learning examples is fixed for all the experiments so the results reported by the different evaluation approaches is comparable. All experiments are run on an Intel Core i5 3.10 GHz x 4 with 4 GB of RAM.

	$\delta(\Xi, \mathcal{O})$	$\delta(\Xi, *)$	$1 - \frac{\delta(\Xi,\mathcal{O})}{\delta(\Xi,*)}$
blocks	0	90	1.0
driverlog	5	144	0.97
ferry	2	69	0.97
floortile	34	342	0.90
grid	42	153	0.73
gripper	2	30	0.93
hanoi	1	63	0.98
hiking	69	174	0.60
miconic	3	72	0.96
npuzzle	2	24	0.92
parking	5	111	0.95
satellite	24	75	0.68
transport	4	78	0.95
visitall	2	24	0.92
zenotravel	3	63	0.95

Table 1: Evaluation of the quality of the learned models with respect to an observations test set.

MADAGASCAR is the classical planner we use to solve the instances that result from our compilations because its ability to deal with dead-ends [Rintanen, 2014]. In addition, because its SAT-based nature, MADAGASCAR can apply the actions for programming preconditions in a single planning step (in parallel) because these actions do not interact. Actions for programming action effects can also be applied in a single planning step reducing significantly the planning horizon.

The compilation source code, the evaluation scripts and the benchmarks (including the learning and test sets) are fully available at this anonymous repository *https://github.com/anonsub/observations-learning* so any experimental data reported in the paper can be reproduced.

Evaluating with a test set

When a reference model is not available, the evaluation of the learning process can be done with respect to a test set. Table 1 summarizes the obtained results when evaluating the quality of the learned models with respect to an observation test set. The table reports, for each domain, the *observation edit distance* (which is computed with our extended compilation), the *maximum edit distance*, and their ratio.

The reported results showed that, despite we are learning only from 25 state observations, twelve out of fifteen learned domains achieved ratios of 90% or above. This fact evidences that the learned models required small edition amounts to match the observations given in the test set.

Evaluating with a reference model

Here we evaluate the learned models with respect to the actual generative model since, opposite to what usually happens in ML, this model is available when learning classical planning models from the IPC. For each domain, the learned model is compared with the actual model and its quality is quantified with the *precision* and *recall* metrics:

- $Precision = \frac{tp}{tp+fp}$, where tp is the number of *true positives* (predicates that correctly appear in the action

	Pre		Add		Del			
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
blocks	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.44
driverlog	0.0	0.0	0.25	0.43	0.0	0.0	0.08	0.14
ferry	1.0	0.71	1.0	1.0	1.0	1.0	1.0	0.9
floortile	0.38	0.55	0.4	0.18	0.56	0.45	0.44	0.39
grid	0.5	0.47	0.33	0.29	0.25	0.29	0.36	0.35
gripper	0.83	0.83	0.75	0.75	0.75	0.75	0.78	0.78
hanoi	0.5	0.25	0.5	0.5	0.0	0.0	0.33	0.25
hiking	0.43	0.43	0.5	0.35	0.44	0.47	0.46	0.42
miconic	0.6	0.33	0.33	0.25	0.33	0.33	0.42	0.31
npuzzle	0.33	0.33	0.0	0.0	0.0	0.0	0.11	0.11
parking	0.25	0.21	0.0	0.0	0.0	0.0	0.08	0.07
satellite	0.6	0.21	0.8	0.8	1.0	0.5	0.8	0.5
transport	1.0	0.3	0.8	0.8	1.0	0.6	0.93	0.57
visitall	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
zenotravel	0.67	0.29	0.33	0.29	0.33	0.14	0.44	0.24

Table 2: Precision and recall values obtained without computing the $f_{P\&R}$ mapping with the reference model.

	Pre		Add		Del			
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
blocks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
driverlog	0.67	0.14	0.33	0.57	0.67	0.29	0.56	0.33
ferry	1.0	0.71	1.0	1.0	1.0	1.0	1.0	0.9
floortile	0.44	0.64	1.0	0.45	0.89	0.73	0.78	0.61
grid	0.63	0.59	0.67	0.57	0.63	0.71	0.64	0.62
gripper	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hanoi	1.0	0.5	1.0	1.0	1.0	1.0	1.0	0.83
hiking	0.78	0.6	0.93	0.82	0.88	0.88	0.87	0.77
miconic	0.8	0.44	1.0	0.75	1.0	1.0	0.93	0.73
npuzzle	0.67	0.67	1.0	1.0	1.0	1.0	0.89	0.89
parking	0.56	0.36	0.5	0.33	0.5	0.33	0.52	0.34
satellite	0.6	0.21	0.8	0.8	1.0	0.5	0.8	0.5
transport	1.0	0.3	1.0	1.0	1.0	0.6	1.0	0.63
visitall	0.67	1.0	1.0	1.0	1.0	1.0	0.89	1.0
zenotravel	1.0	0.43	0.67	0.57	1.0	0.43	0.89	0.48

Table 3: Precision and recall values obtained when computing the $f_{P\&R}$ mapping with the reference model.

model) and fp is the number of *false positives* (predicates in the learned model that should not appear). Gives a notion of *soundness*.

- $Recall = \frac{tp}{tp+fn}$, where fn is the number of *false negatives* (predicates that should appear in the learned model but are missing). Gives a notion of *completeness*.

Table 2 summarizes the obtained results. Precision (**P**) and recall (**R**) are computed separately for the preconditions (**Pre**), positive effects (**Add**) and negative Effects (**Del**) while the last two columns report averages values.

The learning scores reported in Table 2 are poorer than in Table 1. *Precision* and *recall*, given its syntax nature, report low scores for learned models that are actually good but correspond to *reformulations* of the actual model (changes of the roles of actions or their parameters).

Precision and recall robust to model reformulations

To give insight to the actual quality of the learned models we define a method for computing *Precision* and *Recall* that is robust to such model reformulations. Precision and recall are often combined using the *harmonic mean*. This expression is called the *F-measure* (or the balanced *F-score*) and is formally defined as $F = 2 \times \frac{Precision \times Recall}{Precision + Recall}$. Given the learned STRIPS action model Ξ and the reference STRIPS action model Ξ^*, the bijective function $f_{P\&R} : \Xi \mapsto \Xi^*$ is the mapping between the learned and the reference model that maximizes the accumulated *F-measure*.

Table 3 shows that significantly higher *precision* and *recall* scores are reported when comparing each learned action scheme $\xi \in \Xi$ with the corresponding reference scheme $f_{P\&R}(\xi) \in \Xi^*$ given by the $f_{P\&R}$ mapping. These results show that in all of the evaluated domains, except *ferry* and *satellite*, learning interchanged the roles of actions (or their parameters) with respect to their role in the reference model.

However in few domains, Table 3 still reports lower scores than Table 1. The explanation is that, how the observations of test sets are generated, affect to the results reported on Table 1. For instance, in the *driverlog* domain the learned model missed learning the action scheme for the `disembark-truck` action because this action was not considered in the learning set, the same happened in the *floortile* with the `paint-down` action or in the *parking* domain with the `move-curb-to-curb` action. We realized that these actions do not appear either in the corresponding test sets so the learned action models were not be penalized for not learning these action schemes. The generation of relevant observations of planning domains is still an open research question, planning domains are highly structured which makes that some states likely to have low probability of being chosen by chance [Fern *et al.*, 2004].

6 Conclusions

As far as we know, this is the first work on learning STRIPS action models from state observations, exclusively using classical planning and evaluated over a wide range of different domains. Recently, Stern and Juba 2017 proposed a classical planning compilation for learning action models but following the *finite domain* representation for the state variables and did not report experimental results since the compilation was not implemented. Asai and Fukunaga 2017 proposed an approach for learning PDDL action models on a few domains from large images datasets (thousands of observations per domain).

The empirical results show that since our approach is strongly based on inference can generate good quaility models from very small data sets (ten out of fifteen domains achieved both *Precision* and *Recall* values over 0.75 learning only from 25 observations). Generating *informative* examples for learning planning action models is however an open issue. Planning actions include preconditions that are only satisfied by specific sequences of actions, often, with a low probability of being chosen by chance [Fern *et al.*, 2004]. The success of recent algorithms for exploring planning tasks [Francés *et al.*, 2017] motivates the development of novel techniques able to autonomously collect informative learning examples. The combination of such exploration techniques with our learning approach is an intriguing research direction towards the bootstrapping of planning action models.

References

[Amir and Chang, 2008] Eyal Amir and Allen Chang. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.

[Asai and Fukunaga, 2017] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: From unlabeled images to pddl (and back). *KEPS 2017*, page 27, 2017.

[Bonet and Geffner, 2001] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[Bonet *et al.*, 2009] Blai Bonet, Héctor Palacios, and Héctor Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*, 2009.

[Cresswell *et al.*, 2013] Stephen N Cresswell, Thomas L McCluskey, and Margaret M West. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(02):195–213, 2013.

[Fern *et al.*, 2004] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *ICAPS*, pages 191–199, 2004.

[Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)*, 20:61–124, 2003.

[Francés *et al.*, 2017] Guillem Francés, Miquel Ramírez, Nir Lipovetzky, and Hector Geffner. Purely declarative action representations are overrated: Classical planning with simulators. In *International Joint Conference on Artificial Intelligence*, 2017.

[Geffner and Bonet, 2013] Hector Geffner and Blai Bonet. A concise introduction to models and methods for automated planning, 2013.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[Kambhampati, 2007] Subbarao Kambhampati. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Proceedings of the National Conference on Artificial Intelligence*, 2007.

[McDermott *et al.*, 1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language, 1998.

[Muise, 2016] Christian Muise. Planning. domains. *ICAPS system demonstration*, 2016.

[Ramırez and Geffner, 2009] Miquel Ramırez and Hector Geffner. Plan recognition as planning. In *Proceedings of the 21st international joint conference on Artifical intelligence. Morgan Kaufmann Publishers Inc*, pages 1778–1783, 2009.

[Ramírez, 2012] Miquel Ramírez. *Plan recognition as planning*. PhD thesis, Universitat Pompeu Fabra, 2012.

[Rintanen, 2014] Jussi Rintanen. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 2014.

[Segovia-Aguas *et al.*, 2016] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Hierarchical finite state controllers for generalized planning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 3235–3241. AAAI Press, 2016.

[Segovia-Aguas *et al.*, 2017] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence*, 2017.

[Slaney and Thiébaux, 2001] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[Stern and Juba, 2017] Roni Stern and Brendan Juba. Efficient, safe, and probably approximately complete learning of action models. *IJCAI*, 2017.

[Yang *et al.*, 2007] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2-3):107–143, 2007.

[Zhuo and Kambhampati, 2013] Hankz Hankui Zhuo and Subbarao Kambhampati. Action-model acquisition from noisy plan traces. In *IJCAI*, pages 2444–2450, 2013.