Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 00 (2018) 1–15
Artificial Intelligence

Learning STRIPS action models with classical planning

Diego Aineto[a], Sergio Jiménez Celorrio[a], Eva Onaindia[a]

[a]*Department of Computer Systems and Computation, Universitat Politcnica de Valncia. Spain*

Abstract

This paper presents a novel approach for learning STRIPS action models from observations of plan executions that compiles this learning task into classical planning. The compilation approach is flexible to various amount and forms of available input knowledge; the learning examples can range from a set of plans (with their corresponding initial and final states) or sequences of state observations, to just a set of initial and final states (where no intermediate action or state is given). The compilation accepts also partially specified action models and can be used to validate whether the observation of a plan execution follows a given STRIPS action model, even if this model is not fully specified. What is more, the compilation is extensible to assess how well a STRIPS action model matches observations of plan executions. Last but not least, the paper evaluates the performance of the compilation approach by learning action models for a wide range of classical planning domains from the International Planning Competition (IPC) and assessing the learned models with respect to (1), test sets of observations of plan executions and (2), the true models.

Keywords: Classical planning, Planning and learning, Learning action models, Generalized planning

1. Introduction

Besides *plan synthesis* [1], planning action models are also useful for *plan/goal recognition* [2]. At both planning tasks, an automated planner is required to reason about action models that correctly and completely capture the possible world transitions [3]. 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 [4].

On the other hand, Machine Learning (ML) has shown to be able to compute a wide range of different kinds of models from examples [5]. The application of inductive ML to the learning of STRIPS action models, the vanilla action model for planning [6], is not straightforward though:

- The *input* to ML algorithms (the learning/training data) usually are finite vectors encoding the value of fixed features in a given set of objects. The input for learning planning action models are observations of plan executions (where each plan possibly has a different length).
- The *output* of ML algorithms usually is a scalar value (an integer, in the case of *classification* tasks, or a real value, in the case of *regression* tasks). When learning STRIPS action models the output is, for each action, the sets of preconditions, negative and positive effects, that define the possible state transitions.

Motivated by recent advances on the synthesis of different kinds of generative models with classical planning [7, 8, 9, 10], this paper introduces an innovative approach for learning STRIPS action models that can be defined as a classical planning compilation. A solution to the classical planning task that results from our compilation is a sequence of actions that determines the learned action model, i.e. the preconditions and effects of the target STRIPS operator schemas.

The compilation approach is appealing by itself, because it leverages off-the-shelf planners and because it opens up a way towards *bootstrapping* planning action models, enabling a planner to gradually learn/update its action model. Moreover, the practicality of the compilation allow us to report learning results over a wide range of IPC planning domains. Apart from these, the compilation approach presents the following contributions:

1. *Flexibility in the input knowledge*. The compilation is flexible to various amounts and forms of available input knowledge. Learning examples can range from a set of plans (with their corresponding initial and final states) or state observations, to just a set of initial and final states where no intermediate action or state is observed. The compilation also accepts previous knowledge about the structure of the actions in the form of partially specified action models.
2. *Model Evaluation*. The compilation can assess how well a STRIPS action model matches a given set of observations of plan executions. Our compilation is extensible to accept a learned model as input besides the observations of plan executions. This extension allows us to transform the input model into a new model that induces the observations whilst assessing the amount of edition required by the input model to induce the given observations.

A first description of the compilation and its extensions previously appeared in several conference papers [11]. Compared to the conference papers, the present paper includes the following novel material:

- Unifies the formulation of our approach for learning STRIPS action models from state observations and from labeled plans.
- Shows that our compilation is also valid for the particular scenario where the observed states are not full states but *partial states*, in the sense that some of the observed fluents (either with positive or negative value) are missing.
- Proposes a distance metric to assess how well a STRIPS action model matches a given set labeled plans. Defines the compilation to compute this distance as well as the edit distance between two given STRIPS action models.

Section 2 reviews related work on learning planning action models. Section 3 formalizes the classical planning model with *conditional effects* (a requirement of the proposed compilation) and the STRIPS action model (the output of the addressed learning task). Section 4 formalizes the learning of STRIPS action models with regard to different amounts of available input knowledge. Sections 5 and 6 describe our compilation approach for addressing the formalized learning tasks and how the compilation is extensible to assess learned action models. Section 7 reports the data collected in a two-fold empirical evaluation of our learning approach: First the learned STRIPS action models are tested with a set of state observation sequences and second, the learned models are compared to the corresponding reference model. Finally, Section 8 discusses the strengths and weaknesses of the compilation approach and proposes several opportunities for future research.

2. Related work

Back in the 90's various systems aimed learning operators mostly via interaction with the environment. LIVE captured and formulated observable features of objects and used them to acquire and refine operators [12]. OBSERVER updated preconditions and effects by removing and adding facts, respectively, accordingly to observations [13]. These early works were based on lifting the observed states supported by exploratory plans or external teachers, but none provided a theoretical justification for this second source of knowledge.

More recent work on learning planning action models [14] shows that although learning STRIPS operators from pure interaction with the environment requires an exponential number of samples, access to an external teacher can provide solution traces on demand.

Whilst the aforementioned works deal with full state observability,action model learning has also been studied in domains where there is partial or missing state observability. ARMS works when no partial intermediate state is given. It defines a set of weighted constraints that must hold for the plans to be correct, and solves the weighted propositional satisfiability problem with a MAX-SAT solver [15]. In order to efficiently solve the large MAX-SAT representations, ARMS implements a hill-climbing method that models the actions approximately. SLAF also deals

with partial observability [16]. Given a formula representing the initial belief state, a sequence of executed actions and the corresponding partially observed states, it builds a complete explanation of observations by models of actions through a CNF formula. The learning algorithm updates the formula of the belief state with every action and observation in the sequence. This update makes sure that the new formula represents all the transition relations consistent with the actions and observations. The formula returned at the end includes all consistent models, which can then be retrieved with additional processing.

Unlike the previous approaches, the one described in [17] deals with both missing and noisy predicates in the observations. An action model is first learnt by constructing a set of kernel classifiers which tolerate noise and partial observability and then STRIPSrules are derived from the classifiers' parameters.

LOCM only requires the example plans as input without need for providing information about predicates or states [18]. This makes LOCM be most likely the learning approach that works with the least information possible. The lack of available information is addressed by LOCM by exploiting assumptions about the kind of domain model it has to generate. Particularly, it assumes a domain consists of a collection of objects (sorts) whose defined set of states can be captured by a parameterized Finite State Machine (FSM). The intuitive assumptions of LOCM , like the continuity of object transitions or the association of parameters between consecutive actions in the training sequence, yield a learning model heavily reliant on the kind of domain structure. The inability of LOCM to properly derive domain theories where the state of a sort is subject to different FSMs is later overcome by LOCM2 by forming separate FSMs, each containing a subset of the full transition set for the sort [19]. LOP (LOCM with Optimized Plans [20]), the last contribution of the LOCM family, addresses the problem of inducing static predicates. Because LOCM approaches induce similar models for domains with similar structures, they face problems at generating models for domains that are only distinguished by whether or not they contain static relations (e.g. *blocksworld* and *freecell*). In order to mitigate this drawback, LOP applies a post-processing step after the LOCM analysis which requires additional information about the plans, namely a set of optimal plans to be used in the learning phase.

Compiling the learning of action models into classical planning is a general and flexible approach that allows to accommodate various amounts and kinds of input knowledge and opens up a path for addressing further learning and validation tasks. For instance, the example plans in Π could be replaced or complemented by a set O of sequences of observations (i.e., fully or partial state observations with noisy or missing fluents [21]), so learning tasks $\Lambda = \langle \Psi, \Sigma, O, \Xi_0 \rangle$ could also be addressed. Furthermore, our approach seems extensible to learning other types of generative models (e.g. hierarchical models like HTN or behaviour trees), that can be more appealing than STRIPS models, since using them to compute planning solutions requires less search effort.

3. Background

This section defines the planning model used on this work, Classical planning with conditional effects, and the output of the learning tasks addressed in the paper, a STRIPS action model.

3.1. 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 (without loss of generality, we will 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. An action $a \in A$ is defined with *preconditions*, $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, *positive effects*, $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, and *negative effects* $\mathsf{eff}^-(a) \subseteq \mathcal{L}(F)$. 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 $\theta(s, a) = \{s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)\}$.

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.

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.

3.2. STRIPS *action schemas*

This work addresses the learning of PDDL action schemas that follow the STRIPS requirement [22, 23]. Figure 1 shows the *stack* action schema, coded in PDDL, from a four-operator *blocksworld* [24].

```
(: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*.

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. We define F_v, a new set of fluents s.t. $F \cap F_v = \emptyset$, that results from instantiating Ψ using only the objects in Ω_v and 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 schemas $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$ 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`)`.
- 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, given the set of predicates Ψ and the header of a STRIPS operator schema ξ, we define $F_v(\xi) \subseteq F_v$ as the subset of elements that can appear in the action schema ξ and that confine its 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`)`$\}$ excludes the fluents from F_v that involve v_2 because the action header `pickup(`v_1`)` contains the single parameter v_1. Given the set of predicates Ψ and the header of the operator schema ξ, the upper-bound of the number of possible STRIPS action model for ξ, is $2^{3|F_v(\xi)|}$.

4. Learning STRIPS action models

Learning STRIPS action models from fully available input knowledge, i.e. from plans where the *pre-* and *post-states* of every action in a plan are available, is straightforward. When any intermediate state is available, 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 pre-states of the corresponding actions [25]. This section formalizes a set of more challenging action model learning tasks, where less input knowledge is available.

4.1. Learning from state observations

This 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 formalized as $\Lambda = \langle \mathcal{M}, \Psi, O \rangle$:

- $\mathcal{M}$ is the set of *empty* operator schemas, wherein each $\xi \in \mathcal{M}$ is only composed of $head(\xi)$. In some cases, we may not require to start learning from scratch that is, the operator schemas in $\mathcal{M}$ may be not *empty* but partially specified operator schemes where some preconditions and effects are a priori known.
- Ψ is the set of predicates that define the abstract state space of a given planning domain.
- $O = \langle s_0, s_1, \ldots, s_n \rangle$ is a sequence of *state observations* obtained observing the execution of an *unobserved* plan $\pi = \langle a_1, \ldots, a_n \rangle$ such that, for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. Let us also define the learning task $\Lambda = \langle \mathcal{M}, \Psi, \sigma \rangle$ where the observed states are just the state pair $\sigma = (s_0, s_n)$, that we call *label*, and that comprises the *final* state s_n resulting from executing the *unobserved* plan π starting from the *initial* state s_0.

A solution to the $\Lambda = \langle \mathcal{M}, \Psi, O \rangle$ learning task is a set of operator schema $\mathcal{M}'$ compliant with the headers in $\mathcal{M}$, the predicates Ψ, and the state observation sequence O. In this learning scenario, a solution must not only determine a possible STRIPS action model but also the plan π, that explain the given observations using the learned STRIPS model. Figure 2 shows a $\Lambda = \langle \mathcal{M}, \Psi, O \rangle$ task for learning the *blocksworld* STRIPS action model from the five-state observations sequence that corresponds to inverting a 2-block tower.

4.2. Learning from a labeled plan

Here we redefine the task of learning STRIPS action models from observations of plan executions to cover the case where the actions executed by the observed agent are known. In this case, the learning task is formalized as $\Lambda = \langle \mathcal{M}, \Psi, \sigma, \pi \rangle$, where:

- The plan $\pi = \langle a_1, \ldots, a_n \rangle$, is an action sequence applicable in $s_0 \in \sigma$ and that generates the state $s_n \in \sigma$.

Figure 3 shows an example of a *blocksworld* learning task $\Lambda = \langle \mathcal{M}, \Psi, \sigma, \pi \rangle$, that corresponds to observing the execution of an eight-action plan for inverting a four-block tower.

4.3. Learning from multiple plans

The previous definitions formalize the learning STRIPS action models from observations of a single plan execution. These definitions are extensible to the more general case where learning from the execution of multiple plans:

- When learning from states observations, $\Lambda = \langle \mathcal{M}, \Psi, O \rangle$ is still a valid formalization but considering that now that $O = \langle s_0^1, s_1^1, \ldots, s_n^1, \ldots, s_0^t, s_1^t, \ldots, s_n^t \ldots, s_0^\tau, s_1^\tau, \ldots, s_n^\tau \rangle$ are sequences of *state observations* obtained observing the execution of the corresponding *unobserved* plan $\pi^t = \langle a_1^t, \ldots, a_n^t \rangle$, $1 \leq t \leq \tau$ such that, for each $1 \leq i \leq n$, a_i^t is applicable in s_{i-1}^t and generates the successor state $s_i^t = \theta(s_{i-1}^t, a_i^t)$.
- When learning from a set of labeled plans the task is defined as $\Lambda = \langle \mathcal{M}, \Psi, \Sigma, \Pi \rangle$, where $\Pi = \{\pi_1, \ldots, \pi_\tau\}$ is a given set of example plans and $\Sigma = \{\sigma_1, \ldots, \sigma_\tau\}$ is the corresponding set of labels such that each plan $\pi_t = \langle a_1^t, \ldots, a_n^t \rangle$, $1 \leq t \leq \tau$, is an action sequence applicable in $s_0^t \in \sigma_t$ and that generates the state $s_n^t \in \sigma_t$.

```
;;;;;; Headers in M

(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 B) (on B A) (ontable A) (handempty)

;;; observation #1
(holding B) (clear A) (ontable A)

;;; observation #2
(clear A) (ontable A) (clear B) (ontable B) (handempty)

;;; observation #3
(holding A) (clear B) (ontable B)

;;; observation #4
(clear A) (on blockA B) (ontable B) (handempty)
```

Figure 2: Example of a $\Lambda = \langle \mathcal{M}, \Psi, O \rangle$ task for learning a STRIPS action model in the *blocksworld* from a sequence of five state observations.

```
;;; Predicates in Ψ

(handempty) (holding ?o  - object)
(clear ?o - object) (ontable ?o - object)
(on ?o1 - object ?o2 - object)
```

```
;;; Plan π

0: (unstack A B)
1: (putdown A)
2: (unstack B C)
3: (stack B A)
4: (unstack C D)
5: (stack C B)
6: (pickup D)
7: (stack D C)
```

;;; Label $\sigma = (s_0^1, s_n^1)$

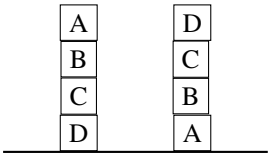

Figure 3: Example of a $\Lambda = \langle \mathcal{M}, \Psi, \sigma, \pi \rangle$ task for learning a STRIPS action model in the blocksworld from a labeled plan.

5. Learning STRIPS action models with classical planning

Our approach for addressing a Λ learning task is compiling it into a classical planning task P_Λ with conditional effects. A planning compilation is a suitable approach for addressing Λ because a solution must not only determine the STRIPS action model $\mathcal{M}'$ but also, the *unobserved* plans that explain the inputs to the learning task. The intuition behind the compilation is that a solution to the resulting classical planning task is a sequence of actions that:

1. **Programs the action model $\mathcal{M}'$.** A solution plan starts with a *prefix* that, for each $\xi \in \mathcal{M}$, determines which fluents $f \in F_v(\xi)$ belong to its $pre(\xi)$, $del(\xi)$ and $add(\xi)$ sets.

2. **Validates the action model $\mathcal{M}'$.** The solution plan continues with a postfix that reproduces the given input knowledge (the observations of the plan executions) using the programmed action model $\mathcal{M}'$.

5.1. Learning from state observations

Given a learning task $\Lambda = \langle \mathcal{M}, \Psi, O \rangle$ 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 O, i.e. `block1` and `block2` in Figure 2.
  - 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/positive effect in the schema $\xi \in \mathcal{M}'$. For instance, the preconditions of the *stack* schema (Figure 1) are represented by the pair of fluents `pre_holding_stack_`v_1 and `pre_clear_stack_`v_2 set to *True*.
  - The fluents $mode_{prog}$ and $mode_{val}$ indicating whether the operator schemas are programmed or validated and the 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 schemas are programmed with every possible precondition (the most specific learning hypothesis), 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.
- $G_\Lambda = \bigcup_{1 \leq i \leq n} \{test_i\}$, requires 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 $\xi \in \mathcal{M}$:
     - Actions for **removing** a *precondition* $f \in F_v(\xi)$ from the action schema $\xi \in \mathcal{M}$.

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

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

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

  2. Actions for *applying* a programmed operator schema $\xi \in \mathcal{M}$ bound with objects $\omega \subseteq \Omega^{ar(\xi)}$. Given that the operators headers are known, the variables $pars(\xi)$ are bound to the objects in ω that appear at the same position.

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

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

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

Known preconditions and effects (that is a partially specified STRIPS action model) can be encoded as fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ set to true at the initial state I_Λ. The corresponding programming actions, $\mathsf{programPre}_{f,\xi}$ and $\mathsf{programEff}_{f,\xi}$, become unnecessary and are removed from A_Λ making the classical planning task P_Λ easier to be solved. In the extreme, when a fully specified STRIPS action model Ξ is given in Ξ_0, the compilation validates whether an observed plan follows the given model. In this case, if a solution plan is found to P_Λ, it means that the given STRIPS action model is *valid* for the given examples. If P_Λ is unsolvable it means that the given STRIPS action model is invalid since it is not compliant with all the given examples.

To illustrate how our compilation works, Figure 4 shows a plan that solves a classical planning task resulting from the compilation. The plan programs and validates the *stack* schema using the five state observations shown in Figure 2 as well as previously specified operator schemes for *pickup*, *putdown* and *unstack*. Note that this compilation is valid for the particular scenario where the observed states are not full states but *partial states*, in the sense that some of the observed fluents (either with positive or negative value) are missing. Even more, the compilation is valid for learning when there are missing state observation in O by removing any reference to the $mode_{val}$ fluent. In that case the classical planner will determine how many *apply* actions are necessary between two *validate* actions.

```
00 :  (program_pre_clear_stack_v1)
01 :  (program_pre_handempty_stack)
02 :  (program_pre_holding_stack_v2)
03 :  (program_pre_on_stack_v1_v1)
04 :  (program_pre_on_stack_v1_v2)
05 :  (program_pre_on_stack_v2_v1)
06 :  (program_pre_on_stack_v2_v2)
07 :  (program_pre_ontable_stack_v1)
08 :  (program_pre_ontable_stack_v2)
09 :  (program_eff_clear_stack_v1)
10 :  (program_eff_clear_stack_v2)
11 :  (program_eff_handempty_stack)
12 :  (program_eff_holding_stack_v1)
13 :  (program_eff_on_stack_v1_v2)
14 :  (apply_unstack block2 block1)
15 :  (validate_1)
16 :  (apply_putdown block2)
17 :  (validate_2)
18 :  (apply_pickup block1)
19 :  (validate_3)
20 :  (apply_stack block1 block2)
21 :  (validate_4)
```

Figure 4: Plan for programming and validating the *stack* schema using the five state observations shown in Figure 2 as well as previously specified operator schemes for *pickup*, *putdown* and *unstack*.

5.2. Learning from a labeled plan

We extend the compilation to consider labeled plans. Given a learning task $\Lambda = \langle \mathcal{M}, \Psi, \sigma, \pi \rangle$, the compilation outputs a classical planning task $P_{\Lambda'} = \langle F_{\Lambda'}, A_{\Lambda'}, I_{\Lambda'}, G_{\Lambda'} \rangle$ such that:

- $F_{\Lambda'}$ extends F_Λ with $F_\pi = \{plan(name(\xi), \Omega^{ar(\xi)}, j)\}$, the fluents to code the steps of plan $\pi = \langle a_1, \ldots, a_n \rangle$. Fluents at_j and $next_{j,j+1}$, $1 \leq j < n$, are also added to represent the current plan step and to iterate through the steps of the plan.
- $I_{\Lambda'}$ extends I_Λ with fluents F_π plus fluents at_1 and $\{next_{j,j+1}\}$, $1 \leq j < n$, for indicating the plan step where the action model is validated. Goals are $G_{\Lambda'} = \{test_1\}$, as now the programmed action model is only validated in the single state observation s_n, the last state reached by the plan. If more intermediate observed states are known they can however be used for validation adding a $validate_i$ action for each observation $s_i \in O$.
- With respect to $A_{\Lambda'}$.
  1. The actions for *programming* the preconditions/effects of a given operator schema $\xi \in \Xi$ and the actions for *validating* the programmed action model in the given state observations are the same.
  2. The actions for *applying* an already programmed operator have an extra precondition $f \in F_\pi$ that encodes the current plan step, and extra conditional effects $\{at_j\} \rhd \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$ for advancing to the next plan step. With this mechanism we ensure that these actions are applied in the same order as in the example plan π.

To illustrate this, the classical plan of Figure 5 is a solution to a learning task $\Lambda = \langle \mathcal{M}, \Psi, \sigma, \pi \rangle$ for getting the *blocksworld* action model where operator schemes for `pickup`, `putdown` and `unstack` are specified in $\mathcal{M}$. This plan programs and validates the operator schema `stack` from *blocksworld*, using the plan π and label σ shown in Figure 3. Plan steps $[0, 8]$ program the preconditions of the `stack` operator, steps $[9, 13]$ program the operator effects and steps $[14, 22]$ validate the programmed operators following the plan π shown in the Figure 3.

```
00 :  (program_pre_clear_stack_v1)
01 :  (program_pre_handempty_stack)
02 :  (program_pre_holding_stack_v2)
03 :  (program_pre_on_stack_v1_v1)
04 :  (program_pre_on_stack_v1_v2)
05 :  (program_pre_on_stack_v2_v1)
06 :  (program_pre_on_stack_v2_v2)
07 :  (program_pre_ontable_stack_v1)
08 :  (program_pre_ontable_stack_v2)
09 :  (program_eff_clear_stack_v1)
10 :  (program_eff_clear_stack_v2)
11 :  (program_eff_handempty_stack)
12 :  (program_eff_holding_stack_v1)
13 :  (program_eff_on_stack_v1_v2)
14 :  (apply_unstack a b i1 i2)
15 :  (apply_putdown a i2 i3)
16 :  (apply_unstack b c i3 i4)
17 :  (apply_stack b a i4 i5)
18 :  (apply_unstack c d i5 i6)
19 :  (apply_stack c b i6 i7)
20 :  (apply_pickup d i7 i8)
21 :  (apply_stack d c i8 i9)
22 :  (validate_1)
```

Figure 5: Plan for programming and validating the *stack* schema using plan π and label σ (shown in Figure 3) as well as previously specified operator schemes for *pickup*, *putdown* and *unstack*.

Addressing the learning task $\Lambda = \langle \mathcal{M}, \Psi, \Sigma, \Pi \rangle$ with classical planning requires introducing a small modification to our compilation. In particular, the actions for *validating* the labeled plan $\pi_t \in \Pi$ with the corresponding label

$\sigma_t \in \Sigma$, $1 \leq t \leq \tau$ are now defined as:

$$\mathsf{pre}(\mathsf{validate_t}) = G_t \cup \{test_j\}_{1 \leq j < t} \cup \{\neg test_j\}_{t \leq j \leq \tau} \cup \{\neg mode_{prog}\},$$
$$\mathsf{cond}(\mathsf{validate_t}) = \{\emptyset\} \rhd \{test_t\} \cup \{\neg f\}_{\forall f \in G_t, f \notin I_{t+1}} \cup \{f\}_{\forall f \in I_{t+1}, f \notin G_t}.$$

5.3. Compilation properties

Lemma 1. *Soundness. Any classical plan π that solves P_Λ induces an action model $\mathcal{M}'$ that solves $\Lambda = \langle \mathcal{M}, \Psi, O/\Sigma, \Pi \rangle$.*

Proof sketch. Once operator schemas $\mathcal{M}'$ are programmed, they can only be applied and validated, because of the $mode_{prog}$ fluent. In addition, 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 schemas that reaches every state $s_i \in O$, starting from s_0 and following the sequence $1 \leq i \leq n$. This means that the programmed action model $\mathcal{M}'$ complies with the provided input knowledge and hence, solves Λ. □

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

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

The size of the classical planning task P_Λ output by the compilation depends on:

- The arity of the actions headers in $\mathcal{M}$ and the predicates Ψ that are given as input to the Λ learning task. The larger these numbers, the larger the $F_v(\xi)$ sets, that define the $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ fluents set and the corresponding set of *programming* actions.
- The number of given state observations. The larger $|O|$, the more $test_i$ fluents and $\mathsf{validate_i}$ actions in P_Λ.

5.4. Exploiting static predicates to optimize the compilation

A *static predicate* $p \in \Psi$ is a predicate that does not appear in the effects of any action [26]. Therefore, one can get rid of the mechanism for programming these predicates in the effects of any action schema while keeping the compilation complete. Given a static predicate p:

- Fluents $del_f(\xi)$ and $add_f(\xi)$, such that $f \in F_v$ is an instantiation of the static predicate p in the set of *variable objects* Ω_v, can be discarded for every $\xi \in \Xi$.
- Actions $\mathsf{programEff_{f,\xi}}$ (s.t. $f \in F_v$ is an instantiation of p in Ω_v) can also be discarded for every $\xi \in \Xi$.

Static predicates can also constrain the space of possible preconditions by looking at the given set of labels Σ. One can assume that if a precondition $f \in F_v$ (s.t. $f \in F_v$ is an instantiation of a static predicate in Ω_v) is not compliant with the labels in Σ then, fluents $pre_f(\xi)$ and actions $\mathsf{programPre_{f,\xi}}$ can be discarded for every $\xi \in \Xi$. For instance in the *zenotravel* domain *pre_next_board_v1_v1*, *pre_next_debark_v1_v1*, *pre_next_fly_v1_v1*, *pre_next_zoom_v1_v1*, *pre_next_refuel_v1_v1* can be discarded (and their corresponding programming actions) because a precondition `(next ?v1 ?v1 - flevel)` will never hold at any state in Σ.

Looking at the given example plans, fluents $pre_f(\xi)$ and actions $\mathsf{programPre_{f,\xi}}$ are discardable for every $\xi \in \Xi$ if a precondition $f \in F_v$ (s.t. $f \in F_v$ is an instantiation of a static predicate in Ω_v) is not possible according to Π. Back to the *zenotravel* domain, if an example plan $\pi_t \in \Pi$ contains the action `(fly plane1 city2 city0 fl3 fl2)` and the corresponding label $\sigma_t \in \Sigma$ contains the static literal `(next fl2 fl3)` but does not contain `(next fl2 fl2)`, `(next fl3 fl3)` or `(next fl3 fl2)` the only possible precondition including the static predicate is *pre_next_fly_v5_v4*.

6. Evaluating STRIPS action models

We assess how well a STRIPS action model $\mathcal{M}$ explains a given sequence of observations O according to the amount of *edition* required by $\mathcal{M}$ to induce O.

6.1. Edition of STRIPS action models

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

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

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 *edit operations* have the same positive cost.

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

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 $\xi \in \mathcal{M}$ 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 is twice $|F_v(\xi)|$.

Definition 4. *The* ***maximum edit distance*** *of an* STRIPS *action model $\mathcal{M}$ built from the set of possible elements F_v is $\delta(\mathcal{M}, *) = \sum_{\xi \in \mathcal{M}} 3|F_v(\xi)|$.*

6.2. The STRIPS edit distance

We define now an edit distance to asses the quality of a learned action model with respect to a sequence of state observations.

Definition 5. *Given an action model $\mathcal{M}$, built from F_v, and the observations sequence $O = \langle s_0, s_1, \ldots, s_n \rangle$. The* **observation edit distance***, denoted by $\delta(\mathcal{M}, O)$, is the minimal edit distance from $\mathcal{M}$ to any model $\mathcal{M}'$, defined also within F_v, such that $\mathcal{M}'$ can produce a plan $\pi = \langle a_1, \ldots, a_n \rangle$ that induces O;*

$$\delta(\mathcal{M}, O) = \min_{\forall \mathcal{M}' \to O} \delta(\mathcal{M}, \mathcal{M}')$$

The Λ learning tasks can swap the roles of two operators whose headers match or two action parameters that belong to the same type (e.g. the *blocksworld* operator `stack` can be *learned* with the preconditions and effects of the `unstack` operator and vice versa, or the parameters of the `stack` operator can be swapped). The *observation edit distance* is a semantic measure that is robust to role changes of this kind.

Unlike the error function defined by ARMS [15], the *observation edit distance* assesses, with a single expression, the flaws in the preconditions and effects of a given learned model. This fact enables the recognition of STRIPS action models. The idea, taken from *plan recognition as planning* [27], is to map distances into likelihoods. The *observation edit distance* could be mapped into a likelihood with the following expression $P(O|\mathcal{M}) = 1 - \frac{\delta(\mathcal{M},O)}{\delta(\mathcal{M},*)}$.

The edit distance can also be defined with respect to a labeled plan.

Definition 6. *Given an action model $\mathcal{M}$, built from F_v, and the plan $\pi = \langle a_1, \ldots, a_n \rangle$ with the corresponding label $\sigma = (s_0, s_n)$. The* **plan edit distance***, denoted by $\delta(\mathcal{M}, \pi, \sigma)$, is the minimal edit distance from $\mathcal{M}$ to any model $\mathcal{M}'$, defined also within F_v, such that $\mathcal{M}'$ can produce a plan π that induces σ;*

$$\delta(\mathcal{M}, O) = \min_{\forall \mathcal{M}' \to \sigma} \delta(\mathcal{M}, \mathcal{M}')$$

Last but not least, the edit distance can also be defined with respect to multiple plans.

6.3. Computing the edit distance

Our compilation is extensible to compute the *edit distance* between two models. A solution to the planning task resulting from the extended compilation is a sequence of actions that edits the action model $\mathcal{M}$ to produce $\mathcal{M}'$ using to the two *edit operations*.

The output of the extended compilation is 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 and $mode_{prog}$ set to true. In addition, the input action model $\mathcal{M}$ is now encoded in the initial state. This means that the fluents $pre_f(\xi)/del_f(\xi)/add_f(\xi)$, $f \in F_v(\xi)$, hold in the initial state iff they appear in $\mathcal{M}$.
- A'_Λ, comprises the same three kinds of actions of A_Λ. The actions for *applying* an already programmed operator schema and the actions for *validating* an observation are defined exactly as in the previous compilation. The only difference here is that the actions for *programming* the operator schema now implement the two *edit operations* (i.e. include actions for *inserting* a precondition and for *deleting* a negative/positive effect).

To illustrate this, the plan of Figure **??** solves the classical planning task P'_Λ that corresponds to editing a *blocksworld* action model where the positive effects `(handempty)` and `(clear ?v1)` of the `stack` schema are missing. The plan edits first the `stack` schema, *inserting* these two positive effects, and then validates the edited action model in the five-observation sequence of Figure **??**.

Our interest when computing the *observation edit distance* is not in $\mathcal{M}'$ but in the number of required *edit operations*, e.g. $\delta(\mathcal{M}, \mathcal{O}) = 2$ for the example in Figure 6. The *observation edit distance* is exactly computed if P'_Λ is optimally solved (according to the number of edit actions); is approximated if P'_Λ is solved with a satisfying planner (our case); and is a less accurate estimate (but faster to be computed) if the solved task is a relaxation of P'_Λ [28].

```
00 :  (insert_add_handempty_stack)
01 :  (insert_add_clear_stack_var1)
```

Figure 6: Plan for editing a given *blockswold* schema and validating it at the state observations shown in Figure **??**.

6.4. Computing the observations distance

Our compilation is extensible to compute the *observation edit distance* by simply considering that the model $\mathcal{M}$, given in $\Lambda = \langle \mathcal{M}, \Psi, \mathcal{O} \rangle$, is *non-empty*. In other words, now $\mathcal{M}$ is a set of given operator schemas, wherein each $\xi \in \mathcal{M}$ initially contains $head(\xi)$ but also the $pre(\xi)$, $del(\xi)$ and $add(\xi)$ sets. A solution to the planning task resulting from the extended compilation is a sequence of actions that:

1. **Edits the action model $\mathcal{M}$.** A solution plan starts with a *prefix* that modifies the preconditions and effects of the action schemas in $\mathcal{M}$ using to the two *edit operations*.
2. **Validates the edited model $\mathcal{M}'$ in $\mathcal{O}$.** The solution plan continues with a postfix that validates the edited model on the given observations $\mathcal{O}$, as in Section 4.

Now Λ is not a learning task but the task of editing $\mathcal{M}$ to produce the observations $\mathcal{O}$, which results in the edited model $\mathcal{M}'$. The output of the extended compilation is 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 and $mode_{prog}$ set to true. In addition, the input action model $\mathcal{M}$ is now encoded in the initial state. This means that the fluents $pre_f(\xi)/del_f(\xi)/add_f(\xi)$, $f \in F_v(\xi)$, hold in the initial state iff they appear in $\mathcal{M}$.
- A'_Λ, comprises the same three kinds of actions of A_Λ. The actions for *applying* an already programmed operator schema and the actions for *validating* an observation are defined exactly as in the previous compilation. The only difference here is that the actions for *programming* the operator schema now implement the two *edit operations* (i.e. include actions for *inserting* a precondition and for *deleting* a negative/positive effect).

To illustrate this, the plan of Figure **??** solves the classical planning task P'_Λ that corresponds to editing a *blocksworld* action model where the positive effects `(handempty)` and `(clear ?v1)` of the `stack` schema are missing. The plan edits first the `stack` schema, *inserting* these two positive effects, and then validates the edited action model in the five-observation sequence of Figure **??**.

Our interest when computing the *observation edit distance* is not in $\mathcal{M}'$ but in the number of required *edit operations* for that $\mathcal{M}'$ is validated in the given observations, e.g. $\delta(\mathcal{M}, O) = 2$ for the example in Figure 7. The *observation edit distance* is exactly computed if P'_Λ is optimally solved (according to the number of edit actions); is approximated if P'_Λ is solved with a satisfying planner (our case); and is a less accurate estimate (but faster to be computed) if the solved task is a relaxation of P'_Λ [28].

```
00 :  (insert_add_handempty_stack)
01 :  (insert_add_clear_stack_var1)
02 :  (apply_unstack block2 block1)
03 :  (validate_1)
04 :  (apply_putdown block2)
05 :  (validate_2)
06 :  (apply_pickup block1)
07 :  (validate_3)
08 :  (apply_stack block1 block2)
09 :  (validate_4)
```

Figure 7: Plan for editing a given *blockswold* schema and validating it at the state observations shown in Figure **??**.

7. Recognition of STRIPS action models

8. Evaluation

This section evaluates the performance of our approach for learning STRIPS action models starting from different amounts of available input knowledge.

8.1. Setup

The domains used in the evaluation are IPC domains that satisfy the STRIPS requirement [23], taken from the PLANNING.DOMAINS repository [29]. We only use 5 learning examples for each domain and they are fixed for all the experiments so we can evaluate the impact of the input knowledge in the quality of the learned models. All experiments are run on an Intel Core i5 3.10 GHz x 4 with 4 GB of RAM.

- **Reproducibility**. We make fully available the compilation source code, the evaluation scripts and the used benchmarks at this anonymous repository *https://github.com/anonsub/strips-learning* so any experimental data reported in the paper is fully reproducible.

- **Planner**. The classical planner we use to solve the instances that result from our compilations is MADAGASCAR [30]. We use MADAGASCAR because its ability to deal with planning instances populated with dead-ends. In addition, SAT-based planners 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.

- **Metrics**.The quality of the learned models is quantified with the *precision* and *recall* metrics. These two metrics are frequently used in *pattern recognition*, *information retrieval* and *binary classification* and are more informative that simply counting the number of errors in the learned model or computing the *symmetric difference* between the learned and the reference model [31]. Intuitively, precision gives a notion of *soundness* while recall gives a notion of the *completeness* of the learned models. Formally, $Precision = \frac{tp}{tp+fp}$, where tp is the

number of true positives (predicates that correctly appear in the action model) and fp is the number of false positives (predicates appear in the learned action model that should not appear). Recall is formally defined as $Recall = \frac{tp}{tp+fn}$ where fn is the number of false negatives (predicates that should appear in the learned action model but are missing).

9. Conclusions

Acknowledgment

This work is supported by the Spanish MINECO project TIN2017-88476-C2-1-R. Diego Aineto is partially supported by the *FPU16/03184* and Sergio Jiménez by the *RYC15/18009*, both programs funded by the Spanish government.

References

[1] M. Ghallab, D. Nau, P. Traverso, Automated Planning: theory and practice, Elsevier, 2004.
[2] M. Ramírez, Plan recognition as planning, Ph.D. thesis, Universitat Pompeu Fabra (2012).
[3] H. Geffner, B. Bonet, A concise introduction to models and methods for automated planning (2013).
[4] S. Kambhampati, Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models, in: National Conference on Artificial Intelligence, AAAI-07, 2007.
[5] R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Machine learning: An artificial intelligence approach, Springer Science & Business Media, 2013.
[6] R. E. Fikes, N. J. Nilsson, Strips: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (3-4) (1971) 189–208.
[7] B. Bonet, H. Palacios, H. Geffner, Automatic derivation of memoryless policies and finite-state controllers using classical planners., in: International Conference on Automated Planning and Scheduling (ICAPS), 2009.
[8] J. Segovia, S. Jiménez, A. Jonsson, Generalized planning with procedural domain control knowledge, 2016.
[9] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Hierarchical finite state controllers for generalized planning, in: International Joint Conference on Artificial Intelligence, IJCAI-16, AAAI Press, 2016, pp. 3235–3241.
[10] J. Segovia-Aguas, S. Jiménez, A. Jonsson, Generating context-free grammars using classical planning, in: International Joint Conference on Artificial Intelligence, ICAPS-17, 2017.
[11] D. Aineto, S. Jiménez, E. Onaindia, Learning strips action models with classical planning.
[12] W. Shen, H. A. Simon, Rule creation and rule learning through environmental exploration, in: International Joint Conference on Artificial Intelligence, IJCAI-89, 1989, pp. 675–680.
[13] X. Wang, Learning by observation and practice: An incremental approach for planning operator acquisition, in: International Conference on Machine Learning, 1995, pp. 549–557.
[14] T. J. Walsh, M. L. Littman, Efficient learning of action schemas and web-service descriptions, in: National Conference on Artificial Intelligence, 2008, pp. 714–719.
[15] Q. Yang, K. Wu, Y. Jiang, Learning action models from plan examples using weighted max-sat, Artificial Intelligence 171 (2-3) (2007) 107–143.
[16] E. Amir, A. Chang, Learning partially observable deterministic action models, Journal of Artificial Intelligence Research 33 (2008) 349–402.
[17] K. Mourão, L. S. Zettlemoyer, R. P. A. Petrick, M. Steedman, Learning STRIPS operators from noisy and incomplete observations, in: Conference on Uncertainty in Artificial Intelligence (UAI), 2012, pp. 614–623.
[18] S. N. Cresswell, T. L. McCluskey, M. M. West, Acquiring planning domain models using LOCM, The Knowledge Engineering Review 28 (02) (2013) 195–213.
[19] S. Cresswell, P. Gregory, Generalised domain model acquisition from action traces, in: International Conference on Automated Planning and Scheduling, ICAPS-11, 2011.
[20] P. Gregory, S. Cresswell, Domain model acquisition in the presence of static relations in the LOP system., in: International Conference on Automated Planning and Scheduling, ICAPS-15, 2015, pp. 97–105.
[21] S. Sohrabi, A. V. Riabov, O. Udrea, Plan recognition as planning revisited, in: International Joint Conference on Artificial Intelligence, IJCAI-16, 2016, pp. 3258–3264.
[22] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL – The Planning Domain Definition Language (1998).
[23] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains., Journal of Artificial Intelligence Research 20 (2003) 61–124.
[24] J. Slaney, S. Thiébaux, Blocks world revisited, Artificial Intelligence 125 (1-2) (2001) 119–153.
[25] S. Jiménez, T. De La Rosa, S. Fernández, F. Fernández, D. Borrajo, A review of machine learning for automated planning, The Knowledge Engineering Review 27 (04) (2012) 433–467.
[26] M. Fox, D. Long, The automatic inference of state invariants in TIM, Journal of Artificial Intelligence Research 9 (1998) 367–421.
[27] M. Ramírez, H. Geffner, Plan recognition as planning, in: International Joint conference on Artifical Intelligence, 2009, pp. 1778–1783.
[28] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1-2) (2001) 5–33.
[29] C. Muise, Planning. domains, ICAPS system demonstration.
[30] J. Rintanen, Madagascar: Scalable planning with sat, Proceedings of the 8th International Planning Competition (IPC-2014).

[31] J. Davis, M. Goadrich, The relationship between precision-recall and ROC curves, in: International Conference on Machine learning, ACM, 2006, pp. 233–240.